Operating System Stability and Security through Process Homeostasis

by

Anil Buntwal Somayaji

B.S., Massachusetts Institute of Technology, 1994

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July 2002

Dedication

To all those system administrators who spend their days nursemaiding our computers.

Acknowledgments

Graduate school has turned out to be a long, difficult, but rewarding journey for me, and I would not have made it this far without the help of many, many people. I cannot hope to completely document how these influences have helped me grow; nevertheless, I must try and acknowledge those who have accompanied me.

First, I would like to thank my committee: David Ackley, Rodney Brooks, Barney Maccabe, Margo Seltzer, and especially my advisor, Stephanie Forrest. They all had to read through a rougher manuscript than I would have liked, and had to do so under significant time pressures. They provided invaluable feedback on my ideas and experiments. Most importantly, they have been my teachers.

I'm also grateful to have been part of the Adaptive Computation Group at the University of New Mexico and to have been a member of the complex adaptive systems community here in New Mexico. Fellow graduate students at UNM and colleagues at the Santa Fe Institute have shaped my thoughts and have inspired me to do my best work. I will miss all of you.

The Computer Science Department at UNM has been my home for the past several years. Its faculty members have instructed me and have treated me as a colleague; its administrative staff have helped me with the practical details of being a graduate student. The department has been a supportive environment, and it will be hard to leave. Also, the Systems Support Group of the UNM Computer Science department helped me collect some of my most important datasets. In the process, they sometimes had to deal with crises that I inadvertently created. They have been both helpful and understanding.

Over the past several years many people in the computer security community have helped us understand important past work, yet have also encouraged us to pursue our own vision. They have been valuable colleagues and friends.

The Massachusetts Institute of Technology was my home during my undergraduate years, and I was also a visiting graduate student in its Artificial Intelligence laboratory during the 1996-97 academic year. This year was a valuable part of my graduate school education.

With the help of my former physics teacher, Bill Rodriguez, the University School of Nashville's web server provided the best data on the behavior of pH.

Finally, I must thank my many friends for supporting me, Dana for being there during the last stages of my graduate career, and my parents and my sister Shalini for providing the love and support to get me this far.

This work was funded by NSF, DARPA, ONR, Intel, and IBM.

Operating System

Stability and Security through Process Homeostasis

by

Anil Buntwal Somayaji

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July 2002

Operating System

Stability and Security through Process Homeostasis

by

Anil Buntwal Somayaji

B.S., Massachusetts Institute of Technology, 1994

Ph.D., Computer Science, University of New Mexico, 2002

Abstract

Modern computer systems are plagued with stability and security problems: applications lose data, web servers are hacked, and systems crash under heavy load. Many of these problems arise from rare program behaviors. pH (process Homeostasis) is a Linux 2.2 kernel extension which detects unusual program behavior and responds by slowing down that behavior. Inspired by the homeostatic mechanisms organisms use to stabilize their internal environment, pH detects changes in program behavior by observing changes in short sequences of system calls. When pH determines that a process is behaving unusually, it responds by slowing down that process's system calls. If the anomaly corresponds to a security violation, delays often stop attacks before they can do damage. Delays also give users time to decide whether further actions are warranted.

My dissertation describes the rationale, design, and behavior of pH. Experimental results are reported which show that pH effectively captures the normal behavior of a variety of programs under normal use conditions. This captured behavior allows it to detect anomalies with a low rate of false positives (as low as 1 user intervention every five days). Data are presented that show pH responds effectively and autonomously to buffer overflows, trojan code, and kernel security flaws. pH can also help administrators by detecting newly-introduced configuration errors. At the same time, pH is extremely lightweight: it incurs a general performance penalty of only a few percent, a slowdown that is imperceptible in practice.

The pH prototype is licensed under the GNU General Public License and is available for download at http://www.cs.unm.edu/∼soma/pH/.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Self-awareness is one of the fundamental properties of life. In humans this property is commonly identified with consciousness; however, all living systems are self-aware in that they detect and respond to changes in their internal state. In contrast, computer systems routinely ignore changes in their behavior. Unusual program behavior often leads to data corruption, program crashes, and security violations; despite these problems, current computer systems have no general-purpose mechanism for detecting and responding to such anomalies.

This dissertation is a first step towards fixing this shortcoming. In this and subsequent chapters, a prototype system called pH ("process Homeostasis") is described and evaluated. pH can detect and respond to changes in program behavior and is particularly good at responding to security violations. pH is also an example of a new approach to system design, one that better reflects the structure of biological systems.

The rest of this chapter explains the motivations for pH, describes its basic design, and summarizes the contributions of this dissertation. The final section outlines subsequent chapters.

1.1 Motivation

As our computer systems have become increasingly complex, they have also become more unpredictable and unreliable. Today we routinely run dozens if not hundreds of programs on any given computer. Many of these executables require tens of megabytes of memory and hundreds of megabytes of disk space. As our systems become faster and larger, programs continue to expand in size and complexity.

Although these programs contain a remarkable amount of functionality, the additional capabilities have exacted a correspondingly large cost in reliability and security. New vulnerabilities are found almost every day on most major computer platforms. Even worse, we have all become inured to program quirks and outright crashes. No sooner does one version seem to stabilize, than a new one comes out providing new capabilities and new problems.

One of the main drivers of this rise in complexity is the need to be connected, both in local area networks and on the Internet. Web browsers and chat clients continuously communicate with the outside world, peer-to-peer file sharing services turn workstations into servers, and even word-processors have become Internet-aware, able to transfer documents to and from web servers with a mouse click. Thus, the complexity of flexible user-friendly software is compounded by the need to interact with unpredictable outside data sources. Where an isolated system might behave consistently over time, a networked system is a new system every day, as the rest of the Internet changes around it.

These two factors, complexity and connectivity, together conspire to undermine our trust in our computers. Fundamentally, there is too much going on for even the most sophisticated user to keep track of. If we cannot keep track of what our systems are doing, our computers must monitor and take care of themselves.

1.2 Computer Homeostasis

From its inception, computer science has conceptualized a computer as a machine that is capable of executing any program it is given. This point of view is the essence of the Universal Turing machine and is also the basis of the general-purpose personal computer. In practice, however, any given computer is not used in its full generality; instead, it is repeatedly used for specific tasks (generally on a daily basis), and only occasionally is it called upon to run new programs.

Even though devices ranging from large, multi-processor servers to hand-held Personal Digital Assistants (PDAs) typically run a small set of applications, outside of the embedded realm computers are built to be general-purpose. Hardware makes minimal distinctions between different kinds of programs, and operating systems allocate most resources relatively fairly: memory and network bandwidth are given out on a first-come-first-served basis (until pre-specified limits are reached), and schedulers ensure that every process receives some CPU time on a regular basis. This fairness is good if we assume that no program is more important than any other; if we assume that users prefer some programs and some program states, though, this fairness can be harmful.

Most computer users would like a system to function consistently over time, so that if the computer worked properly when first installed, it would continue to do so in the future. Given that such consistency cannot be guaranteed, it would be nice to know when new circumstances cause a change in system behavior — especially if that behavior could cause problems. We can visualize this distinction by considering the diagram in Figure 1.1. Most of the time a program's behavior is confined to the inner circle of normal behavior. A program wanders out of this region and into that of legal program behavior when unusual events such as communication failures or invalid input data cause normally-unused code to be executed.

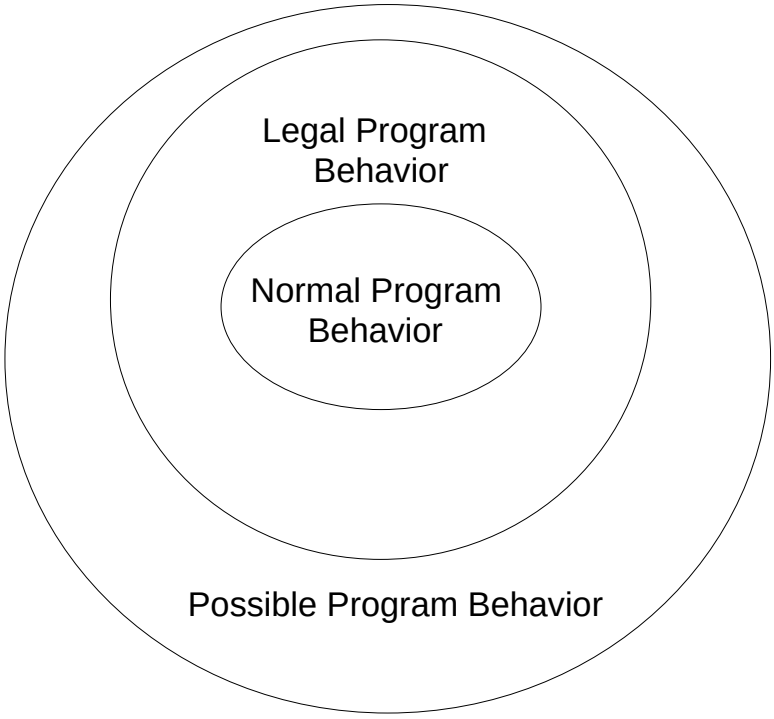

Figure 1.1: A schematic diagram of classes of computer behavior. Note that a program may behave in ways not specified in the source because of inserted foreign code (buffer overflows), compiler errors, hardware faults, and other factors.

If hardware never executed code incorrectly, and if our programs perfectly accounted for all possible errors and interactions, all legal program behavior would be permissible and safe, and no other behaviors would be possible. In practice, though, our programs are far from perfect, and the interactions of these imperfections can cause any program to behave like any other program, for example turning a word processor into a file deletion utility. Such unexpected functionality can lead to security violations and data loss.

One way to handle dangerous program behavior is to create detectors to recognize these states explicitly. This approach is used by most misuse intrusion detection systems, usually through the scanning of data streams for attack signatures. Although

it can be effective, this strategy has two basic limitations. The first is that in general it is impossible to specify all of the ways in which programs can malfunction, and so we are left with systems that must be continually updated as new threats arise. In addition, a dangerous pattern of activity in one context may be perfectly valid in another; thus, signatures in software products must be created for the lowest common denominator to minimize false alarms.

My approach, however, has been to recognize that most programs work correctly most of the time, and so normal program behavior is almost always safe program behavior. By learning normal program behavior through observing a system, and by using this knowledge to detect and respond to unusual, potentially dangerous program behavior, we can improve the stability and security of our computer systems.

People have had relatively little experience in building systems that can detect and respond appropriately to unexpected anomalies; nature, however, has had billions of years of experience. As explained in Chapter 3, living systems employ a variety of detectors and effectors to maintain *homeostasis*, or a stable internal environment. Whether driven by the need to destroy invading organisms or the need for a consistent temperature and chemical environment, these systems constantly monitor the state of an organism and react to perturbations.

The fundamental idea of this dissertation is that we can improve the stability of computer systems by adding homeostatic mechanisms to them. Practical computer users often do their utmost to avoid program upgrades, precisely to avoid changing their stable computational environment. By analogy with biology, a homeostatic operating system would take this idea further and would dynamically maintain a stable computational environment. A truly homeostatic operating system would be extremely complex, and would interconnect multiple detectors and effectors to stabilize many aspects of a system. As a first step towards this goal, I have focused on developing a simple feedback loop consisting of an abnormal system call detector and

an effector which delays anomalous system calls. As the next section explains, there are technical and aesthetic reasons for these choice; more than anything else, though, they are interesting mechanisms because they lead to a system that can detect and respond to harmful system changes, whether they are caused by a configuration error, misuse of a service, or an outright attack.

1.3 System-call Monitoring and Response

As a first step towards a homeostatic operating system, I have focused on monitoring and response at the UNIX system-call level. This basic approach could be used anywhere that there is a programming interface, and it is possible to build similar systems that work with function calls, object method invocations, or even microprocessor instructions. The system call interface, however, has several special properties that make it a good choice for monitoring program behavior for security violations. What follows is an explanation of what a system call is, followed by a brief summary of how pH performs system-call level anomaly-detection and response.

On UNIX and UNIX-like systems, user programs do not have direct access to hardware resources; instead, one program, called the kernel, runs with full access to the hardware, and regular programs must ask the kernel to perform tasks on their behalf. Running instances of a program are known as *processes*. Multiple processes can execute the same program; even though they may share code, each process has its own virtual memory area and virtual CPU resources. The kernel shares memory and processor time between processes and ensures that they do not interfere with each other.

When a process wants additional memory, or when it wants to access the network, disk, or other I/O devices, it requests these resources from the kernel through *system calls*. Such calls normally takes the form of a software interrupt instruction that

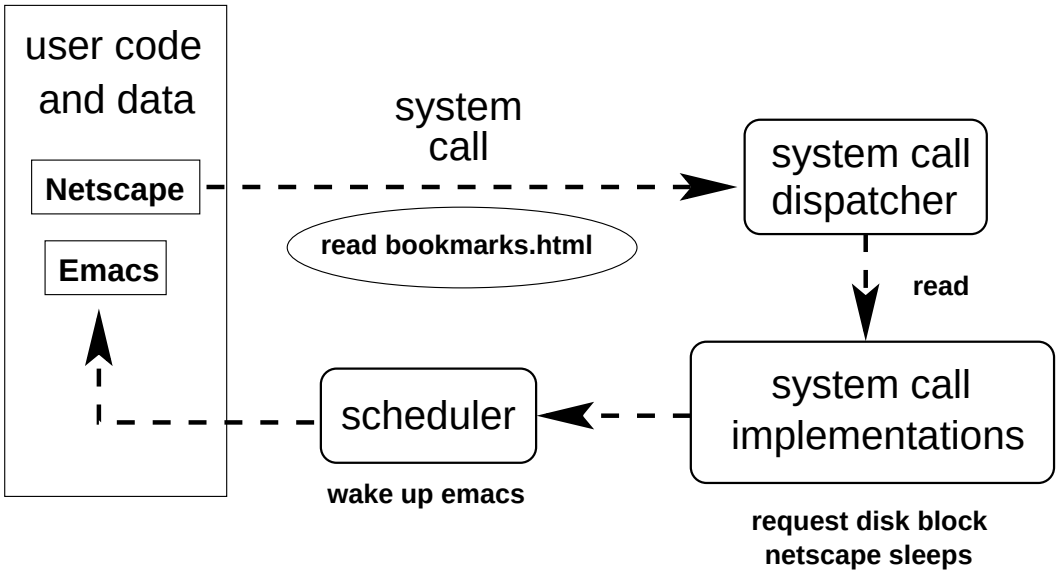

Figure 1.2: The process of making a system call. In this example, a Netscape process makes a read system call to access a user's bookmarks. While netscape is waiting for data from the disk, control is passed to an Emacs process. Note how control moves from user-space to kernel-space and back.

switches the processor into a special supervisor mode and invokes the kernel's system-call dispatch routine. If a requested system call is allowed, the kernel performs the requested operation and then returns control either to the requesting process or to another process that is ready to run. Figure 1.2 shows a visual representation of this transaction between user-space (normal user processes running user programs) and kernel-space.

With the exception of system calls, processes are confined to their own address space. If a process is to damage anything outside of this space, such as other programs, files, or other networked machines, it must do so via the system call interface. Unusual system calls indicate that a process is interacting with the kernel in potentially dangerous ways. Interfering with these calls can help prevent damage; therefore, a computational homeostatic mechanism that monitors and responds to unusual

system calls can help maintain the stability and security of a computer system.

pH is a prototype of such a homeostatic mechanism. It is an extension of the Linux 2.2 kernel that detects and responds to anomalous program behavior as it occurs. It monitors the system calls executed by every process on a single host, maintaining separate profiles for every executable that is run. As each process runs, pH remembers the sequence of recent system calls issued by that process. When a program is first run, if a given sequence has not been executed before by the program, pH remembers a simple generalization of this sequence. Once pH sees no novel system call sequences associated with a given program for a sufficient period of time, it then delays instances of that program that execute novel system call sequences. This delay is exponentially related to the number of recent anomalous system calls; thus, an isolated anomaly produces only a short delay, while a cluster of anomalies cause a process to be effectively halted.

1.4 Contributions

This work makes several concrete contributions. First, it supports the idea presented in our past work [43, 49] that system can be used to detect novel attacks caused by buffer overflows, trojan code, and kernel flaws. System call monitoring can also detect other problems such as configuration errors, helping administrators find problems before a loss of service is reported. Further, it shows that system-call monitoring is practical, in that it can be performed online in real-time with a minimum of overhead.

This work also presents the first graduated approach to automated response in a computer intrusion detection system, namely proportional delays to anomalous system calls. This mechanism can limit the damage caused by novel attacks; in addition, because small system-call delays are barely noticeable, this graduated response helps pH achieve a low rate of false positives, with pH requiring as few as one

user intervention every five days (on average) on a production web server.

Finally, this work demonstrates that system-call monitoring and response can form a key component of a homeostatic operating system.

1.5 Overview

The rest of this dissertation proceeds as follows. Chapter 2 presents related work, situating pH in the fields of operating systems, computer security, and other fields of inquiry. Chapter 3 describes some example homeostatic systems found in living systems and explains and motivates the design of pH. Chapter 4 discusses methods for analyzing system calls.

Chapter 5 discusses the implementation of pH, and presents data on pH's performance overhead. Chapter 6 presents results showing how pH performs on a few normally behaving computers. Data are presented which compare lookahead pairs and sequences. This chapter also examines rates of false-positives, and analyzes the diversity of profiles between different programs and computers. Chapter 7 addresses the issue of detecting and responding to usage anomalies, configuration errors, and real security violations. Anomalies are traced back to the program code that generated them, and it is shown that pH detects novel code paths. pH is also shown to be capable of stopping both abuse and attacks. Chapter 8 summarizes the contributions of this work, analyzes its shortcomings, and presents ideas for future work.

Chapter 2

Background

Although my research has been inspired and informed by work from several fields, pH is perhaps best classified as an "anomaly intrusion detection and response" system, and as such, it should be compared to other work in this computer security sub-field. My work also bears some similarity to work in operating systems, fault-tolerance, system administration, artificial life, and robotics. In the following sections, I discuss how my work relates to these fields of inquiry.

2.1 Computer Security

The field of computer security ultimately deals with the problem of unauthorized or dangerous computer access. Covered within this umbrella are issues of data integrity, confidentiality, and system availability. When dealing with these problems, there are three basic approaches one can take:

- Build systems in such a way that dangerous or unauthorized activities cannot take place.

- Detect such activities as they happen, and ideally stop them before systems are compromised.
- Detect systems that have been compromised, and determine how much damage has been done.

Clearly the first option is preferable, and before the rise of the Internet this was the main focus of the field. Military organizations in particular funded the development of provably secure systems. One famous document produced by the US Department of Defense, known as the Orange Book [84], enshrined various requirements for different levels of trust. By trust, they meant three things: how much you could believe that your system was secure; if it was compromised, how much the system would limit damage; and, after security violation, how well you could trust the logs of the system to tell you what damage had been done. Several trusted operating systems have been built and major UNIX vendors, such as Hewlett-Packard and Sun have trusted (B1-certified) products available [81]. These offerings are generally expensive and hard to administer, and no operating system certified at the B1 level or higher has been successful outside of niche markets. Instead, the broader market has been dominated by systems that are inexpensive, fast, flexible, and feature-rich. Since modern, widely-deployed computer operating systems and applications have well-known fundamental security limitations, users have turned to add-on programs to enhance security.

Some of these additions stop attacks before they can succeed. For example, network firewalls [22] and the TCP Wrappers package [113] restrict network connections in an attempt to exclude dangerous machines and services. Vulnerability scanners such as SATAN [42] and Nessus [36] search for known vulnerabilities on a host or network, allowing them to be proactively fixed. Packages such as StackGuard [32] and the Openwall Linux Kernel Patch [37] prevent many kinds of buffer-overflow attacks from succeeding, either by killing programs that experience stack corruption,

or by preventing the foreign stack-resident code from running.

Other additions detect damage after it has occurred. Packages such as Tripwire [59] detect changes to system files by maintaining a set of cryptographically-secure checksums that are periodically recalculated and verified. Virus-protection software such as Norton AntiVirus [108] scan local or network storage for signatures of known viruses, allowing users the opportunity to either clean or delete infected files[1].

Systems such as pH which detect attacks as they happen are known as *intrusion detection* systems.

2.1.1 Intrusion Detection

The field of intrusion detection dates back to Anderson's 1980 technical report [3] and Denning's 1987 intrusion detection model [35]. Early work in this field was motivated by the needs of government and the military; the need for better security on the growing Internet, however, has led to numerous research projects and commercial systems. Although there is no set of widely accepted rigorous classifications of intrusion detection systems, they can be broadly classified by what level they monitor (host-based or network), and by how they detect attacks (signature, specification, or anomaly). Within this framework, pH is a host-based anomaly intrusion detection system. Before describing other such systems it is worthwhile to consider other types of intrusion detection systems. (For a more complete overview, see Bace [8]; for other recent taxonomies of the intrusion detection field, see Hervè et al. [34] and Axelsson [7].)

Since intruders often attack a system through its connections with the outside

[1]Modern virus-protection packages also scan web pages and email messages for viruses, and can detect when a program attempts to modify executables (a common method for virus propagation). When acting in this mode, these programs are also acting as intrusion-detection systems.

world, a natural approach to intrusion detection is to monitor network traffic and services. Because of the complexity of network traffic, much attention has been devoted to the problem of what parts of the stream should be examined. One interesting approach is taken by the Network Security Monitor, which focuses on characterizing the normal behavior of a network by examining the source, destination, and service of TCP packets [47]. This work served as the basis for Hofmeyr's work on LISYS, a distributed anomaly network intrusion detection system [50].

In contrast, commercial network intrusion detection systems such as the Cisco Secure Intrusion Detection System [24] and ISS RealSecure [52] are signature-based systems, in that they scan packets headers and payloads searching for attack patterns defined by hand-crafted matching rules. The primary advantages of signature-based systems is that they can detect known attacks immediately upon deployment (unlike anomaly-based systems), and they do not need detailed information on the behavior of applications (unlike specification-based systems). The downside is that these systems require frequent signature updates to recognize new attacks, and signatures developed in the laboratory may generate unexpected false positives in the real world.

Host-based intrusion detection systems potentially can use many different data sources. Rather than design custom tools to observe system behavior, most early research in host-based intrusion detection focused on the use of data from audit packages. Audit packages record events such as authorization requests, program invocations, and (some) system calls. This voluminous data is generally written to a special-purpose binary log file. Care is taken to record the data in a secure fashion so that unauthorized users cannot easily conceal their activities. Audit trails are designed to provide forensic evidence for human analysts; however, they can also provide a basis for an automated intrusion detection system. Some of the most sophisticated uses of audit trails were the IDES and NIDES projects, which used SunOS Basic Security Module (BSM) audit data and statistical models to look for

unusual patterns of user behavior [71]. Audit packages by themselves tend to be costly in terms of system performance and storage requirements, and packages such as NIDES only add to the burden. As a result, audit-based intrusion-detection systems have not been widely fielded outside of government agencies[2].

Although audit data has been a popular data source, many other sources have been used. Products such as the ISS RealSecure OS Sensor [53] and the free *logcheck* package [89] detect intrusions by scanning standard system logs for attack patterns. Kuperman [64] developed a technique for generating application-level audit data through library interposition. Zamboni [119], using an "internal sensors" approach, modified the applications and kernel of an OpenBSD system to report a variety of attempted attacks and other suspicious activity.

One noteworthy product that uses a similar framework to pH is the the CylantSecure intrusion detection system [105]. Rather than observing system calls, it uses a heavily modified Linux kernel to detect anomalously-behaving programs. Published papers [38, 80] indicate that Cylant's technology can be used to instrument the source code of arbitrary programs to report when different "modules" are entered and exited. In the CylantSecure product, the behavior of an instrumented Linux kernel is fed into a statistical model which then detects dangerous program behavior and network connections by observing unusual patterns of kernel behavior caused by those programs and connections. CylantSecure appears to be similar in spirit to pH in that it gathers data at the kernel level and performs anomaly detection. It is difficult to make a detailed comparison, though, because little has been published on the algorithms or performance of the system, particularly with respect to false-positive

[2]I considered whether I wanted to use an audit package as my data source; unfortunately, I found that they were cumbersome to use, recorded data that I did not care about (authorization events, network activity), and most importantly recorded events after they had occurred. An automated response mechanism based on audit data would generally be triggered after anomalous events had already occurred rather than while they were happening. In contrast, I wanted pH to be able to respond as anomalies were generated.

rates.

Other groups have chosen to use system calls to detect security violations. Ko et al. [61] formally specified allowable system calls and parameters for privileged programs. More recently, Wagner and Dean created a two-part prototype system that would dynamically monitor the system calls of a program based on a pre-computed static model derived from the program's source. An anomaly is noted when the program makes a system call that the source would not permit [114]. This approach is effective in detecting two of the most common forms of attack, namely buffer overflows and trojan code not present in normal binaries; however, it cannot detect attacks that require only existing code.

Several researchers have built on our original work with system-call sequences [43], developing a number of related, but different approaches. Some have focused on applying the sequence technique to other data streams. Stillerman et al. [107] used sequences of CORBA method invocations for anomaly detection. Jones and Lin [56] used sequences of library calls to detect attacks. Others have tried developing better techniques for detecting anomalies in system call data. Lee et al. [65] used RIPPER, a rule inference system. Marceau [74] developed a method for capturing system-call patterns using a variable-length window. Michael and Ghosh [78] developed and analyzed two finite-state machine analysis techniques, and Ghosh et al. [45] compared sequence analysis with a feed-forward backpropagation neural network and an Elman network. Some researchers have varied both the data source and the analysis method. Endler [39] trained a neural network to detect anomalies in sequences of Solaris BSM audit data. Jones and Li [55] examined "temporal signatures" constructed from system-call sequences augmented with inter-call timing information. Our work has also helped inspired some more theoretical studies. Maxion & Tan [77] compared sequence and Markov anomaly detectors using artificial data sets, focusing on their coverage of possible anomalies. Separately, they have also set forth a set of criteria

for comparing anomaly detection systems [76].

One of the most promising recent innovations was made recently by Sekar et al. [99]. They developed a technique for inferring a finite-state machine using system-call data along with the corresponding program counter values. They claimed that their method converges much faster than the sequence method; however, because of the complications introduced by dynamic linking, their tests effectively ignored the structure of system calls made by library functions. Because the sequence method recorded this information, the comparison was somewhat unfair. Nevertheless, the method is promising enough that it deserves a more careful study.

2.1.2 Automated Response

While detecting attacks is important, it is only half the problem: the other half is what to do once an attack has been detected. The obvious solution is to stop the attack, perhaps by terminating the offending program or network connection. Commercial systems such as Cisco Secure Intrusion Detection System [24] and ISS RealSecure [52] do offer optional responses such as increased logging, termination of offending processes or network connections, or even the ability to block hosts or networks. Because false positives can require unacceptable amounts of administrator attention and cause degradations in service, though, these responses are normally disabled.

One approach to solving this problem is to say that security policies should be precise enough so that there aren't any (interesting) false positives. Systems that restrict the behavior of untrusted helper applications (Janus [46]), foreign code (Java Virtual Machines [67], Xenoservers [94]), and privileged programs [61, 100, 101] all fall in this category. Although this strategy can improve security, it does not remove the need for intrusion detection: indeed, these systems are vulnerable to exactly the

same problems that plague operating systems.

Another approach is to isolate suspicious activities from the rest of the system. Attackers believe that they are changing the state of the system, when they are instead affecting an isolated environment. If the behavior turns out to be legitimate, changes can be propagated to the rest of the system. This technique is particularly applicable to database transactions [70], but can also be applied to regular UNIX services [12, 100, 101]. A related technique is the use of honeypots [25, 109], which are hosts that offer fake services on instrumented hosts. By monitoring the activities of attackers in honeypots, it is possible to learn more about their techniques; further, honeypots trick attackers into wasting resources on useless targets.

One problem with defending against many network attacks is that faked return addresses can mask the identity of the attacker(s). Systems which try and trace network-based attacks to their source [115] can be used to help identify the appropriate target to block, even in the face of faked return addresses.

Some systems address the response problem by having a repertoire of responses combined with control rules. This approach is taken by EMERALD [86], the successor project to NIDES[3]. Ragsdale et al. [93] proposed a framework for adapting responses based upon their usefulness in the past. AngeL [85] is a Linux kernel extension which blocks several host and network-based attacks using situation-specific matching and response rules. Portsentry [90] defends against portscans by blocking connections from suspected remote hosts. Although these systems can be effective against attacks for which they have been designed to respond, they have little ability to respond to newly-encountered attacks.

As discussed earlier, pH slows suspicious activity instead of forbidding it. Although this approach hasn't been used by intrusion detection systems before, it has

[3]The host-based intrusion detection component of EMERALD, eXpert-BSM, has been released [68], but the eResponder expert system component is still in development.

been used by other security mechanisms. Nelson [83] recognized that delays in the form of "unhelpfulness" can be useful security mechanisms. The LaBrea system [69] responds to rapidly-propagating worms such as Code Red and NIMDA by creating virtual machines which accept connections but then disappear. Since attacking machines must wait for the TCP timeout interval before moving onto the next target, the rate of infection is greatly reduced.

Also, delays have long been used on most UNIX systems to enhance security at the login prompt. Typically, there is a few second delay after each failed login attempt, and after a minute the login process times out, forcing a reconnect on remote logins. These delays can be mildly irritating to a clumsy user; however, they also make it much more difficult for a remote attacker to try many different username/password combinations.

2.1.3 Kernel-level Security Projects

The popularity and the open license of the Linux kernel have inspired a variety of security enhancement projects. Projects such as SubDomain [31], SELinux [82], Trustix [6], and LIDS [2] allow fine-grained security policies to be implemented using a variety of techniques and access-control models. AngeL [85] blocks several specific host-based attacks, and also prevents many kinds of network-based attacks on other hosts. Medusa DS9 [120] allows access control to be implemented in a userspace process which can permit, forbid, or modify kernel requests using custom security policies.

Ko et al. [60] have implemented system-call specification, signature, and sequence monitoring on FreeBSD using their software wrappers package. This package (which is also available for Linux, Solaris, and Windows NT) allows one to modify the behavior of system calls using code written in their Wrapper Description Language

(WDL), which is a superset of C. These "wrappers" may be layered, allowing one to compose security policies. It appears that the functionality of pH could be implemented in WDL, although the use of their general framework would probably result in a significantly slower implementation (see Section 5.10).

Engler et al. [41] have developed an interesting and useful tool for checking code correctness. Instead of looking for errors based on pre-defined rules, their checker infers rules based on the "normal" usage of various constructs: it identifies the incorrect use of locks, memory, or other resources by detecting anomalous code patterns. They have applied their system to the Linux and OpenBSD kernels and have found many errors in both programs [23].

2.2 Operating Systems

Although pH is probably best described as an intrusion detection and response system, it can also be seen as a process scheduler that uses an unusual form of feedback. Typical process schedulers choose what process to run based on its static priority, CPU usage, and whether its requested I/O has completed. Other schedulers have incorporated additional information and decision criteria. Fair-share schedulers [48, 58] attempt to allocate CPU time fairly amongst users rather than processes. Massalin and Pu [75] created a fine-grained adaptive scheduler for the Synthesis operating system "analogous to the hardware phased lock loop" that scheduled processes based on timer interrupts, I/O operations, system call traps, and other program behavior statistics. The feedback of pH is much coarser than the Synthesis scheduler, but both make scheduling decisions based on learned patterns of behavior.

Experimental extensible kernel systems, such as SPIN [11, 10] and VINO [40], also have similarities to pH. First, most employ novel OS protection mechanisms such as safe languages or a transaction system to help ensure the safety of grafts inserted

into the kernel. Similarly, pH is a novel OS protection mechanism for Linux, but one that protects the standard system call interface. Also, designers of extensible kernels are interested in gathering data on system behavior and modifying the operation of the system based on this information. Seltzer & Small [102] in particular discuss techniques for having a system monitor and adjust its behavior autonomously. Their focus, however, is on performance, not stability and security.

2.3 Compilers

Most modern compiler environments use some form of "feedback-directed optimization" to improve the quality of complied code. All of these systems use profiles of program behavior to guide program optimization; they differ, however, in how they gather behavior profiles and in when they perform optimizations. Some systems perform optimization and profiling offline (Machine SUIF [103]), some gather profiles online but optimize offline (Morph [121]). Just-in-time compilation environments both profile and optimize online, allowing virtual instruction sets such as Java bytecodes [67] to run as fast as natively-compiled code (Jalapeño [19], HotSpot [79]). This same basic technology can even be used to speed up native binaries (Dynamo [9]) or to allow one processor to masquerade as another (Crusoe [30]).

There are many other systems that perform feedback-directed optimization [104]; what these systems have in common, though, is that they optimize code paths that are run frequently. To do this, these systems must determine how frequently different functions or basic blocks are executed. It should be possible to build a pH-like system that uses this frequency information to directly detect unusual program behavior, and in fact Inoue [51] has recently used Java method invocation frequencies to detect security violations. The primary drawback to using the infrastructure of feedback-directed optimization to detect security violations is that few security-critical systems

currently use this technology. As just-in-time compilation environments become more widespread, it should be possible to use their program behavior-monitoring capabilities to improve system security without reducing system performance.

2.4 Fault Tolerance

Although my work's primary application is to computer security, its goals are similar to those of the field of software fault tolerance. The field of fault tolerance is focused on methods for ensuring reliable computer operation in the face of flawed components, both hardware and software. Most work on fault tolerance has focused on hardware fault tolerance, primarily through the use of redundant components. Software fault tolerance, however, focuses on the problem of flawed software [72]. By assuming that the hardware is reliable, flaws in software must come from flawed specifications, designs, or implementations. These flaws cannot be mitigated through simple replication; instead, software fault tolerant systems use two basic techniques: fault detection and recovery, and design diversity.

When there is only one version of the application available, faults are detected through special error-detection code and are mitigated through exception handlers. Numerous programs employ these techniques in an ad-hoc fashion. In addition, several methodologies for structuring error detection and recovery have been developed, the most prominent of which is the "recovery block" method. A recovery block consists of an acceptance test, followed by several alternatives. Execution of the block proceeds as follows. First, the program is checkpointed. Then, the first alternative is executed, and the acceptance test is run. If the test returns true, the other alternatives are skipped, extra state information is discarded, and the program proceeds. If the test fails, the program is restored to the checkpointed state, the second alternative is run, and the acceptance test is run again. This process continues until the

acceptance test returns true, or until the program runs out of alternatives. Recovery blocks can be nested, allowing for elaborate recovery schemes [72, pp. 1–21]. In the related field of software rejuvenation, applications are periodically reset to a known good state when statistics such as run-time and memory usage reach pre-specified limits. In a clustered environment, nodes may also be reset after first migrating applications to other, redundant nodes [20, 111]. With the increasing use of clusters to run large-scale web and database servers, software rejuvenation has become an effective technique for ensuring system reliability.

Design diversity is based on the hope that independent solutions to a given problem will have different errors, allowing them to be used to check each other. There are two major approaches to design diversity: N-version programming (NVP) and N self-checking programs (NSCP). NVP systems contain N complete implementations of a given specification, each written by different teams of programmers. These versions are run concurrently, generally on different processors, with the output of the system being the output of the majority of the implementations, assuming that a majority agree on a single action. NSCP systems also consist of multiple implementations of a given program specification. However, in NSCP only one of these versions is active at any time, with the others serving as "hot spares." These programs have error-checking code that attempts to detect incorrect behavior. If the active version detects a problem with its behavior, it activates a spare and becomes inactive. The spare implementations may not offer full system functionality, but like a spare tire, they should be enough to keep the system limping along until the problem can be fixed [72, pp. 49–51].

The field of self-stabilization focuses on algorithms which can recover from transient failures automatically. Self-stabilizing algorithms for mutual-exclusion, clock synchronization, and other communications protocols [54] can serve as important building blocks for fault-tolerant distributed systems; however, by potentially mask-

ing the presence of errors, they may cause necessary responses to be delayed.

The spread of the Internet has prompted research into the creation of large-scale fault-tolerant systems. For example, OceanStore [95] uses byzantine protocols, redundant encodings, automated node insertion and removal mechanisms, and other techniques to create a robust, self-maintaining storage infrastructure designed to scale to "billions of users and exabytes of data [95, p. 40]." Phalanx [73] is a software system for building secure, distributed applications such as Public-Key Infrastructure (PKI) and national voting systems through the use of quorum-based protocols. Both of these distributed systems can provide remarkable guarantees of service if we assume that node failures are independent events; if the underlying software is homogeneous, though, such assumptions are not necessarily warranted.

2.5 System Administration

System administrators have long created ad-hoc mechanisms to handle routine maintenance and to detect and respond to potential problems. For example, most UNIX systems automatically rotate log files and flush mail queues. Administrators often add scripts to purge temporary files, perform nightly backups, and to upgrade software packages. Burgess's *cfengine* [16, 17] provides a customizable framework for more elaborate automatic administration systems. *cfengine* periodically runs and checks for conditions such as low available disk space, non-functional daemons, or modified configuration files. Scripts are run when a given trigger condition has been satisfied, solving problems without the need for direct intervention. The design of *cfengine* was also inspired by the human immune system [18]; its knowledge-intensive design, though, is very different from pH's.

Versions of Microsoft Windows [28] starting with Windows 95 have incorporated a number of features to help home users maintain their computers. Windows can

automatically detect and configure new hardware. It notices when its primary disk is nearly full and runs an application to help users delete unnecessary files. Windows can automatically install device drivers and libraries from compressed archives, and applications such as Microsoft Office XP use new Windows services to repair themselves if program files are damaged or deleted [29].

The detection and response mechanisms of *cfengine* and Windows can automate many system administration tasks; when advanced users try to perform maintenance manually, though, these mechanisms can cause significant problems. For example, *cfengine* can undo configuration file changes, preventing quick fixes from being preserved. Windows will often reinstall drivers for devices that have been deliberately removed, sometimes perpetuating hardware conflicts that the user was trying to eliminate. pH is not free of this problem, and Section 6.10 describes how pH can interfere with normal administration tasks. A major challenge for any automated response system is how to prevent normally helpful mechanisms from causing their own problems.

2.6 Artificial Life & Robotics

Although very different in form and functionality, much of the inspiration for pH has come from the field of artificial life, and in particular the work of David Ackley. Through the ccr project [1], Ackley has advocated the view that some existing computational systems are in fact living systems, and if we are to tap the possibilities of our computers, we must at least understand the design principles of biological systems. Although ccr can be described as a peer-to-peer, distributed multi-user dungeon (MUD), it can also be described as an experiment in making computer artifacts that treat communication as a risky endeavor. Communication is inherently dangerous, and this danger is reflected in the structure of all biological systems. Whether you

examine a cell membrane, the human immune system, or predator-prey interactions in ecosystems, it is clear that there is a need to manage resources that are devoted and exposed to interactions with others. Most programs have primitive notions of resource and access control; either an interaction is permitted, or not. And, if that interaction is permitted, all requested resources are granted. In contrast, ccr limits all interactions in a way analogous to trusted operating systems. Movement within ccr is strictly regulated. Communication between worlds is controlled, both in terms of what kinds of information, and how much may pass between worlds. There are bandwidth regulators on all channels, and if one world attempts to flood another with more data than allowed, the excess information is blocked and eventually the connection is terminated. Also, certain information is considered inherently private (such as a world's private key) — such information is carefully managed, and is not allowed to flow over any outside connection, no matter how much trust is attributed to that connection. Such design features have framed my views on how a computer should behave.

While the ccr project has influenced my aesthetics, the field of robotics has informed my views of implementation. In particular, Rodney Brooks's work on subsumption architecture has been inspirational. In a 1985 MIT AI Lab memo, Brooks described what was then a new approach to mobile robot control [13]. Through the use of loosely coupled feedback loops, each connecting a limited set of sensor inputs to a small set of actuators, he was able to have a robot interact with real environments in a rapid and robust fashion. One particular difference with traditional robotics was that there was no centralized representation of the outside world; different components might in fact have contradictory models. However, by arranging these modules in a hierarchy (with higher modules "subsuming" lower ones), and using this hierarchy to arbitrate between conflicting actions, these different (mostly implicit) representations are able collectively to provide robust control for a robot. This approach has been quite fruitful, and has produced a number of successful

robots [14]. Other researchers have adopted the subsumption architecture, using more traditional AI algorithms in the higher levels (such as planners) to produce more sophisticated behaviors [62, pp. 6–12].

Although the field of robotics might appear distant from that of operating systems, with the rise of the Internet both face a common challenge: they must interact with a rapidly changing, potentially threatening outside world on relatively short time scales. Although a networked computer does not have to deal with faulty sensors and imprecise motor control, it does have to deal with a barrage of network packets and a multitude of independent programs and users, some of which may be malicious. In both cases, getting precise and current information about the state of the world is difficult and expensive, whether it be a room full of furniture and people, or a busy web server communicating with machines around the world. Like many robotics systems, pH uses an efficient learning algorithm and a few heuristics to connect simplified inputs to useful actions.

Chapter 3

Homeostasis

The success of biological systems can seem puzzling when considered from the viewpoint of computer security and computer science. Most of the standard tools for producing robust computational systems, namely specification, design, and formal verification, were not used to create most lifeforms; instead, they have evolved over time to survive and reproduce within a variety of environments. This process of reproduction, variation, and selection has produced organisms that have fundamental flaws which leave them vulnerable to disease, old age, starvation, and death; nevertheless, living systems are also remarkably robust, and are able to autonomously survive and reproduce in the most unlikely of circumstances. One fundamental organizing principle that enables this robustness is homeostasis.

This chapter explains homeostasis in living systems and discusses how homeostasis inspired and informed the design of pH. The first section describes two examples of biological homeostasis. The second section outlines the requirements of pH's design, explains how four abstractions of biological homeostasis informed the design of pH, and gives a user's view of pH in action.

3.1 Biological Homeostasis

All biological systems maintain a stable internal state by monitoring and responding to internal and external changes. This self-monitoring is one of the defining properties of life and is known as "homeostasis." By minimizing variations in the internal state of an organism, homeostatic mechanisms help ensure the smooth operation of the numerous chemical and mechanical systems that constitute an organism.

Although many homeostatic mechanisms have been studied extensively, most are still incompletely understood. What follows is a high-level description of two mechanisms, temperature control and the immune system, that are used by the human body to keep us alive. Although our knowledge of both is far from perfect, these examples are still useful as inspiration for analogous computational mechanisms.

3.1.1 Temperature Control

Cells employ numerous enzymes (biological catalysts) to control the chemical reactions necessary for their survival [33, pp. 169–170]. The effectiveness of these enzymes is often influenced by temperature: an enzyme may work well within a narrow temperature range, but may become permanently damaged by being exposed to temperatures significantly outside this range [33, pp. 175–176]. Because malfunctioning enzymes can cause death, living systems have evolved mechanisms to cope with variations in external temperature.

For example, cold-blooded animals such as reptiles regulate their internal temperature by moving to warmer or colder areas as needed. Although this strategy is energy efficient, it means that cold-blooded animals can only be active when their surroundings are warm, i.e. during the daytime. In contrast, warm-blooded animals such as birds and mammals metabolize food to generate heat, allowing them to be

active after dark and to survive in very cold climates. Since most enzymes are efficient only within a narrow temperature range, humans and other warm-blooded animals employ many mechanisms to maintain a constant internal temperature.

The human body detects temperature changes through sensors (specialized nerve cells) in the skin and inside the body. As we become cold, blood vessels in the extremities constrict, reserving a greater proportion of the body's warmth to the inner core. Shivering is induced, causing muscles to produce additional heat. In some animals, feathers or hairs are made to stand up, increasing the amount of air trapped near the skin, enhancing the insulating properties of the outer layers. In humans, however, this same mechanism simply produces goose bumps [33, p. 786]. Extended exposure to the cold causes individuals to eat more food, in response to the increased metabolic demands of heat generation .

Analogous mechanisms happen in response to heat: we sweat, blood vessels in our extremities dilate, and over time we tend to eat less. Note that at the onset of a fever, we have reactions (such as shivering) that are associated with cold. These symptoms occur because our body is attempting to reach a new, higher temperature equilibrium. While very high temperatures (above $40°C$) can cause dementia and convulsions, moderately higher temperatures stimulate the immune system and disrupt the functioning of invaders [112, pp. 588–597]. When a fever breaks, we tend to sweat: the danger has passed, and it is time to move back to a normal temperature equilibrium.

3.1.2 Immune System

The human immune system is a multi-layered, complex system which is responsible for defending the body from foreign pathogens and misbehaving cells. Although it can be seen as analogous to a military defense system, in many ways the immune

system is closer in spirit to the homeostatic mechanisms discussed above. Millions of cells roam our bodies, each attempting to detect specific perturbations in the body's internal environment. Cellular damage, unusual cellular behavior, or just unusual chemical compounds can all cause an immune response. Any immune response must be proportional to the change detected, much as the violence of our shivering depends on how cold we are. Further, the type and severity of an immune response must be balanced against other factors. For example, many immune responses result in the killing of healthy cells; thus, an overzealous immune response can cause more damage than an invading pathogen. In the end, there is never a complete victory for the immune system; rather, success is the continued survival of the body.

The immune system has many different components which interact and regulate each other. Much is known about some of these components, while others are still quite mysterious. Hofmeyr [98] has written a useful introduction to the complexities of the immune system from a computer scientist's perspective. Rather than describing many of these systems, here we focus on one part of the adaptive immune response: the interaction between T-cells and MHC-peptide complexes.

As each cell recycles its proteins, some peptide fragments are not reused immediately. Instead, they are used to inform the immune system about the internal state of the cell. This communication is facilitated by a protein called the major histocompatibility complex, or MHC. MHC has a cleft in its middle, large enough for an 8–10 amino acid peptide fragment to fit in. (See Figure 3.1.) This cleft must be filled with a peptide fragment in order for MHC to be stable; otherwise, it breaks down into its component polypeptides [21, p. 4:8]. Peptide fragments are transported into areas that are topologically outside of the cell (the lumen of the endoplasmic reticulum or intracellular vesicles), where unbound MHC constituents are stored. When a suitable peptide fragment comes into contact with MHC components, a complete MHC is formed. This MHC is then transported to the cell's surface, to be presented

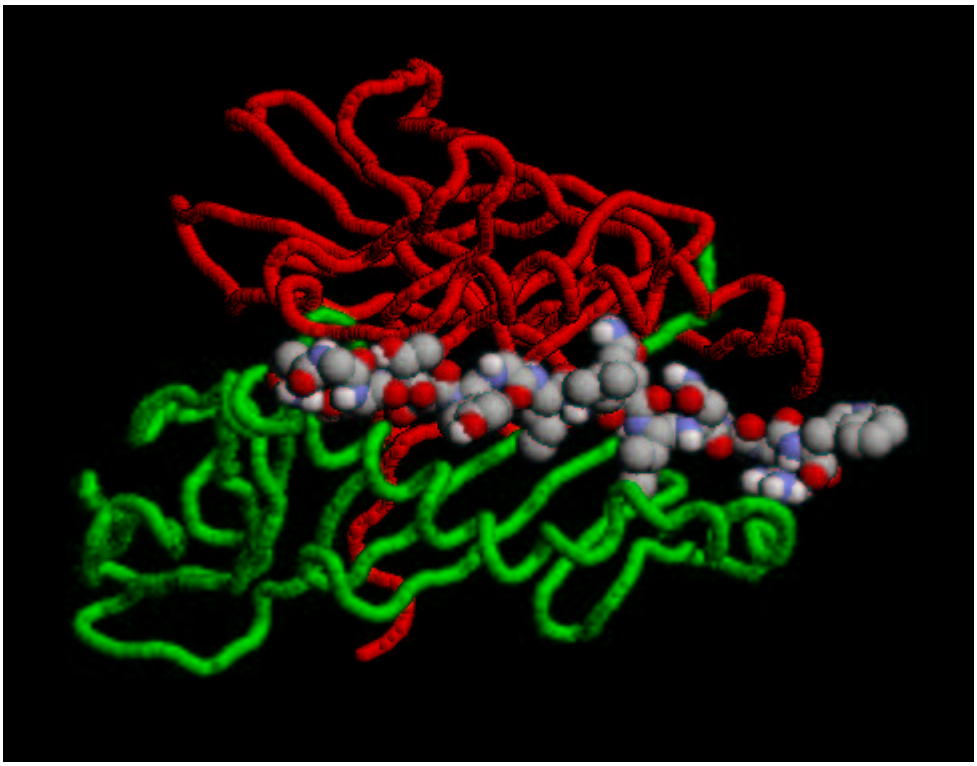

Figure 3.1: A rendering of a MHC molecule with a bound peptide. This diagram shows an MHC class II DQ8 protein bound to a papilloma virus E4 peptide. The red part is the alpha polypeptide chain, and the green is the beta chain. The bound peptide is shown using multi-colored space-filling spheres. [118].

to passing immune system cells called T-cells [21, pp. 4:14–15].

Each T-cell has on its surface receptors that are designed to bind to MHC present on another cell's surface. Different T-cells have different receptors, each specialized to recognize a different MHC/peptide fragment combination. During T-cell development, each T-cell constructs its receptors using a random combinatorial process that can produce 10^{16} different receptors [21, 4:35]. Note that this set of receptors can bind to almost any peptide fragment. However, during T-cell development T-cells that bind to MHCs containing "normal" peptide fragments are killed. Thus, when roaming the body, if a T-cell is able to bind to a cell's MHCs, that cell must be

producing an abnormal protein. As proteins directly control the behavior of a cell, it can be assumed that the target cell is behaving abnormally, either because of a genetic defect or because it has been invaded by a pathogen such as a virus. Therefore a T-cell, upon finding a target cell with MHC it recognizes, can initiate an immune response against that cell and its neighbors, generally resulting in cell death.

3.2 Process Homeostasis

Although are not as complex or robust as biological systems, they do have some mechanisms that can be said to maintain homeostasis. For example, because electronics are sensitive to temperature extremes, computer systems have temperature sensors (thermistors and silicon-embedded diodes) and regulatory mechanisms (heatsinks and fans) analogous to those of the human body. If instead we think about computers on the software rather than the hardware level, we must consider other kinds of "temperature" if we wish to draw biological analogies. In living systems, temperature is regulated because the fundamental operations of life (chemical reactions) depend upon the proper functioning of temperature-dependent enzymes. Similarly, computer programs require resources such as CPU cycles and memory to execute correctly. Hence subsystems that help programs receive sufficient resources, such as virtual memory and process scheduling, are analogous to biological temperature-regulation mechanisms.

In contrast, there are relatively few mechanisms in existing computer systems which are analogous to the immune system. Some computers (particularly corporate desktop machines) can detect when they have been opened; modern operating systems have security mechanisms to regulate access to programs and data; virus scanners and signature-based intrusion-detection systems can detect the presence of known dangerous activities at the program and network levels (see Chapter 2). None

of these systems, however, are anywhere as robust, general, or adaptive as the human immune system.

pH was designed to give computer systems homeostatic capabilities analogous to those of the human immune system. Because the constraints of living and computational systems are very different, however, we cannot create a useful computer security mechanism by merely imitating biology. My approach has been first to choose a set of requirements similar to those of the immune system. I then created abstractions that captured some of the important characteristics of biological homeostatic systems and then used these abstractions to guide my design of pH.

The next section describes the requirements of pH's design and compares these requirements with the characteristics of the human immune system. The next section explains the abstractions used to translate biological homeostasis into process homeostasis. Subsequent sections then describe the rationale for each of pH's instantiations of these abstractions.

3.2.1 Requirements

pH was designed to supplement existing UNIX security mechanisms by detecting and responding to security violations as they occur. It was also designed to be analogous to the human immune system. To fulfill both of these constraints, pH was designed to satisfy the following requirements:

- **Broad Detection:** Much as the immune system can detect the presence of almost any pathogen, pH should detect a wide variety of attacks.

- **Effective Response:** The immune system can prevent most pathogens from killing a person; similarly, pH should stop attacks before they can do damage.

- **No Updates:** Although the immune system performance can be enhanced through vaccinations, in general it can adapt on its own to new threats. To provide capabilities analogous to those of the immune system, then, pH should not require an update every time a new security vulnerability is discovered.
- **Lightweight:** pH should have a minimal impact on system resources, to the point that users do not feel motivated to disable pH to improve the performance of their systems.
- **Minimal Administration:** Since users and administrators are already overburdened with the complexity of current systems, pH should be mostly autonomous, requiring minimal human interaction.
- **Secure:** It should not be easy to circumvent pH, and it should not add (significant) security vulnerabilities.

These are a challenging set of requirements that are not met by existing computer security systems. Because the requirements were inspired by biology, it made sense to look to it for means to satisfy them. The next section explains how biology was used to design pH.

3.2.2 Abstractions

Although biology can be a rich source of inspiration, it can also be misleading. Homeostatic systems work well in a biological context; however, there are vast differences between proteins and silicon chips, and so we cannot assume that the lessons of one domain directly apply to the other. One way to bridge this gap is to recognize that there are underlying organizational similarities, or abstractions, between different homeostatic mechanisms. By identifying and translating each organizational feature into an appropriate computational context, we should be able to avoid inappropriate

Abstraction	Temperature Regulation	MHC Immune Response
enclosed system	human individual	human individual
system property	temperature	safe cellular proteins
detector	specialized nerve cells	MHC on cell surfaces
effector	muscles, sweat glands, others	T-cells, cell death

Table 3.1: Four organizational divisions present in biological homeostatic systems.

Abstraction	process Homeostasis (pH)
enclosed system	individual computer
system property	normal program behavior
detector	sequences of system calls
effector	delayed system calls

Table 3.2: The homeostatic organization of pH.

biological imitations, and instead create computational systems that are useful on their own merits.

To see how this may be accomplished, notice that homeostatic systems generally seem to have the following divisions: an enclosed system that needs to be maintained (typically the interior of the organism), a property that must be monitored for changes, detectors that summarize the state of that property, and effectors which change the state of the monitored property. Table 3.1 shows how these abstractions map onto temperature regulation and the MHC/T-cell response of the human immune system. For a computer mechanism to be homeostatic, then, it should have similar abstractions.

Table 3.2 summarizes how these four abstractions map onto pH. The following sections explain and motivate each of these mappings.

3.2.3 Enclosed System

Living systems all have an internal environment that is distinct from the outside world. Homeostatic mechanisms must maintain the stability and integrity of this internal environment if the organism is to survive. If we are to follow the homeostasis analogy, we must choose the internal environment, or the enclosed system, that is to be maintained by our computational homeostatic mechanism. This enclosed system in effect defines the "individual organism" for the purposes of homeostasis.

Although we could maintain homeostasis within a specially defined boundary, it is simplest to leverage the existing boundaries of a system. There are many potential boundaries to choose from. For example, the shared configuration, trust, and administration of networks within an organization can be thought of as defining a single organism, with firewalls acting as a kind of skin separating an intranet from the outside world. At the other extreme, programs (particularly mobile applications) can be seen as autonomous organisms.

The enclosed system for pH is a single networked Linux computer. This choice leverages existing host-based authentication mechanisms and trust boundaries. Although cells and programs are not completely analogous, there is a similar kind of "shared fate:" if the integrity of a Linux computer is compromised, none of its programs are safe, much as all the cells of an individual are in danger if the individual is infected with a disease.

3.2.4 System Property

Having chosen a system to work with, we next needed to decide what property of that system our mechanism should stabilize. There are many properties of a single Linux host that are worth monitoring if we wish to improve system security.

Rather than looking at users or network traffic, though, pH is designed to preserve normal program behavior. Much as T-cells monitor expressed MHC complexes to ensure that cells aren't producing unusual proteins, pH monitors programs to ensure that they aren't running unusual code segments. This evaluation is done relative to a process's current executable and does not directly account for user or network variations; however, because a program's execution environment affects its behavior, relevant external variables are implicitly included in a program's behavior profile.

What this means is that instead of a-priori restrictions on program behavior, pH merely requires that program usage be consistent. The theory is that on an uncompromised system with effective conventional security mechanisms, security violations will cause programs to behave in unusual ways. By detecting and responding to such unusual behavior, pH stops security violations without having pre-specified knowledge of what those violations are. If this assumption is not true and security violations are part of normal program behavior, then either the machine is already compromised, or the work of legitimate users requires the circumvention of existing security mechanisms. In the former case, the game has already been lost; in the latter, there needs to be a change either in user behavior or in the security policy. Both of these cases are thus outside the scope of pH's design.

3.2.5 Detector

To distinguish between normal and abnormal program behavior, we need a generic mechanism for profiling a program's execution. This method should be lightweight so that multiple programs can be monitored concurrently without a significant loss in performance. It should also not interfere with normal program functionality so that monitored programs will continue to work properly. Most importantly, though, the class of abnormal program behavior defined by this mechanism should contain

many kinds of security violations.

There are several reasons why system calls are a good basis for a normal program behavior detector. As explained in Section 1.3, security violations can be detected at the system call level because programs access outside resources through this interface. System calls are also relatively easy to observe: since all system calls invoke the kernel, we can observe every process on a system by instructing the kernel to report system call events. Standard methods for observing system calls, such as the *ptrace* system call, can be very inefficient and can even interfere with normal program operation. A custom kernel mechanism, though, can observe system calls with minimal overhead and without interfering with normal program semantics.

Having decided to observe system calls, we also need a way to classify their patterns. Here again biology provided inspiration in the form of MHC. To review, the MHC captures aspects of a cell's behavior by transporting small chains of amino acids (peptides) to the surface of a cell where they can be inspected by T-cells. In our original paper [43], we found that short sequences of system calls can be used to distinguish normal from abnormal program behavior, much as short chains of amino acids are used to distinguish normal from abnormal cellular behavior.

There is much more to be said about analyzing short sequences of system calls. Chapter 4 describes the lookahead pair method for analyzing system call sequences, which is the method pH uses to detect abnormal program behavior. Chapter 6 explores how well lookahead pairs characterizes normal program behavior, and Chapter 7 shows that the lookahead pairs method is effective at detecting a variety of anomalies, and explores some reasons why it is so effective.

3.2.6 Effector

Given that system calls can be used to detect dangerous program behavior, we now face the challenge of what to do with this information. Other security systems respond to suspected attacks by increasing the level of logging, alerting an administrator, or by avoiding the problem by deferring the decision to another component (see Section 2.1.2). Instead, I wanted pH to react to anomalous calls immediately and effectively.

One fundamental problem for any sort of anomaly-detection system is that of false positives. No matter how good our analysis technique, we have to assume that some detected anomalies are actually instances of acceptable behavior. Further, because we are monitoring every program on a system, and because we are working at the low level of individual system calls, these false positives may occur often. Consequently, we must anticipate that our response mechanism will be invoked frequently, and that some of these invocations will be erroneous.

The human immune system is also fallible, and it often makes mistakes. Still, because of the redundancy and resilience of the human body, these mistakes generally do not cause problems. Individual cells are disposable and replaceable: individually they can be killed with no consequence, and if too many cells are destroyed, others can be created to replace them. This disposability of components (cells) allows the immune system to employ responses that harm both healthy and diseased cells [21, pp. 7:35–7:36].

Current computer systems are not nearly as redundant as living systems, and they have much less capacity for self-repair. Nonetheless, there is one aspect of computers that is almost always disposable: time. Unless there are hard real-time constraints, minor variations in execution time do not affect the correctness of most computations. If a web server takes one second rather than one-tenth of a second

to service a request, it is still operating correctly (albeit with reduced capacity and performance). Having said that, large variations in time do matter in practice, even if correctness is maintained. Users get impatient with sluggish programs, and in a networked environment significant slowdowns can trigger timeouts which cause connections to be terminated or restarted. Taken to the limit, an extremely slow program is indistinguishable from one that has crashed. Thus, to accommodate for inevitable mistakes in detecting abnormal program behavior, pH responds to anomalous program behavior by delaying system calls in proportion to the number of recently anomalous system calls.

Note that a response based on slowing down system calls can be both safe and effective. If false positives cause infrequent anomalies that trigger small delays, then such false positives will result only in minor changes in system behavior. On the other hand, if true positives cause more frequent anomalies which in turn produce larger delays, then such a response can disrupt an attack in many ways. Sufficiently long delays trigger network timeouts, terminating TCP connections. Delays frustrate attackers with slowed responses. Also, long delays give time for a more elaborate analysis of a situation, either by an administrator or by a slower but more sophisticated anomaly detection system.

3.2.7 The User's View of pH

Before discussing the specifics of pH's algorithms and implementation, it is worth considering how pH appears from a user's perspective. This section explains how the previously mentioned mechanisms are integrated together, and it also provides a conceptual roadmap for the next two chapters.

The first step, installation, is relatively simple for a skilled user or administrator: build and install a patched Linux kernel, and then install an initialization script

and a few utility programs. When the system boots, pH remains inactive until its initialization script runs at the start of the multi-user boot process. Once started, pH monitors the system calls made by every process on the system. Profiles are maintained for each executable that is run, stored on disk in a directory tree that mirrors that of the executables themselves. For example, with the default parameter settings, the profile for `/bin/su` is located in `/var/lib/pH/profiles/bin/su`. The first time *su* is run, this profile is created; on subsequent invocations of *su*, the profile is loaded from disk if it is not already present in memory.

When pH is first started, its behavior is imperceptible except for a few additional kernel log messages. After a fixed period of time (by default a week), pH starts reporting that it has begun "normal monitoring" of various executables, meaning that pH can now detect and respond to abnormal behavior by these programs. Many of these programs are simple shell scripts which are periodically run by *cron* at set time intervals; others are system daemons that run in the background. As it observes additional system calls produced by normal computer usage, pH creates normal profiles for more and more commonly used programs; even so, other than the log messages, pH continues to work quietly in the background.

This all changes when the computer is used for new activities. Actions such as installing a new program, reconfiguring a backup script, or attacking a web server can cause pH to delay processes. Applications on UNIX systems often require multiple programs to work together; consequently, even if the profile for the primary program has not stabilized, pH will often react to unusual behavior by utilities such as *sed* and *stty* which are run indirectly by shell scripts or other programs.

A graphical monitoring program, *pHmon*, can be used to monitor pH's actions. When programs are delayed, its main indicator icon changes colors. A user can then click on this icon to interact with a process list. This list can be used to kill the delayed program or to tell pH that the behavior in question is permissible, in which

case pH will allow the program to continue at full speed. If many unusual tasks need to be performed, pH's responses can be disabled; of course, this also means that pH will no longer provide any protection.

The net effect of pH is a system that is tailored to a particular set of users and uses. The more consistent the behavior of this group, the stronger pH's reaction when that behavior changes. Even though the underlying mechanisms are extremely simple, these reactions are easy to anthropomorphize: users will often comment that pH "doesn't like" some program or activity. I find that I can even predict when pH will dislike one of my actions, much like a dog owner can predict that a stranger at the door will cause a barking fit.

Although the behavior of pH can seem sophisticated and lifelike, the next two chapters show that pH's algorithms and implementation are remarkably simple. Chapter 4 explains how pH performs system call sequence analysis, while Chapter 5 covers the rest of pH's implementation.

Chapter 4

Analyzing System Call Sequences

Inspired by the functioning of MHC in the human body, pH analyses program behavior by observing short sequences of system calls. This chapter describes and analyzes the method used by pH to analyze these sequences. The next section discusses our past work on system call sequences and explains why pH observes system calls. The following section then explains two techniques for creating profiles using system call sequences. The last part analyzes and compares these two methods. This analysis gives some rationales for pH using the lookahead pair method for analyzing system call sequences; Chapter 6 presents data that reinforces the validity of this choice.

4.1 Requirements

A method for classifying program behavior based on system-call usage could measure system calls in many ways. It could compare the timings of different system calls, or their relative frequencies. It could analyze arguments to specific system calls, or could only look at a subset of all possible system calls. As discussed in Section 2.1.1, many others have studied different techniques for using system calls to detect

abnormal program behavior and security violations. In our initial 1996 paper [43], however, we decided to use the simplest approach we could conceive, which was to ignore everything about system calls except for their type and relative order.

Given these constraints, what we needed was a way to compress the traces of system calls into a compact profile that quickly converges to a fixed state given "normal" program behavior, while still allowing one to detect abnormal behavior as sets of patterns that aren't represented in the profile. In addition, this modeling algorithm must be able to capture $\sim$ 200 discrete types of events (the size of the system-call vocabulary); it must also permit fast incremental updates, detect low-frequency anomalous events, and have modest memory and CPU requirements.

Without any other knowledge, it might seem that we would need a sophisticated learning algorithm; program behavior is simple enough, though, that we can make due with extremely simple methods. We have used two techniques, both of which use a fixed-length window to partition a process's system calls into sequences. With the most straightforward technique, which we call the sequence method, a profile of a program's behavior consists of the entire set of sequences produced by that program. With the other technique, known as the lookahead pair method, the pairs formed by the current and a past system call are stored in the program's profile.

The sequence method works surprisingly well in comparison to other, more sophisticated algorithms. Warrender [116] compared the sequence method with several others, including a Hidden Markov Model generator and a rule inference algorithm (RIPPER). By analyzing several data sets with each method, she was able to estimate the false and true positive rates of each method. She also roughly measured the execution time required by each method. These experiments showed that the sequence method was almost as accurate as the best algorithm in any given test, while being much less computationally expensive.

Given these and other published results [49, 44], it would seem reasonable for pH to use the sequence method; however, the lookahead pair method is both very fast and very easy to implement, and as our initial paper showed [43], it is also effective at detecting security violations. The rest of this chapter describes and compares these two methods. The analysis shows that lookahead pairs is better suited to the requirements of pH. Chapter 6 presents some data on both methods that reinforces this choice. It also justifies the use of length nine system call sequences.

4.2 Description

In our past work, we used two methods to analyze system-call traces: *sequences* and *lookahead pairs*. For both methods, we define normal behavior in terms of short sequences of system calls. Conceptually, we take a small fixed size window and slide it over each trace, recording which calls precede the current call within the sliding window.

In the sequences method, we record the literal contents of the fixed window in the profile, and the set of these sequences constitutes the model of normal program behavior. With the lookahead pairs method, we record a set of pairs of system calls, with each pair representing the current call and a preceding call. For a window of size x, we can form $x - 1$ pairs, one for each system call preceding the current one. The collection of unique pairs over all the traces for a single program constitutes the model of normal behavior for the program.

More formally, let

$$\begin{aligned} C &= \text{alphabet of possible system calls} \\ c &= |C| \text{ (190 in Linux 2.2, 221 in Linux 2.4)} \\ T &= t_1, t_2, \ldots, t_\tau | t_i \in C \text{ (the trace)} \\ \tau &= \text{the length of T} \\ w &= \text{the window size, } 1 \leq w \leq \tau \\ P &= \text{a set of patterns associated with } T \text{ and } w \\ &\quad \text{(the profile)} \end{aligned}$$

For the sequences method, the profile P_{seq} is defined as:

$$\begin{aligned} P_{seq} = \{\langle s_i, s_{i+1}, \ldots, s_j \rangle : \; & s_i, s_{i+1}, \ldots, s_j \in C, \\ & 1 \leq i, j \leq \tau, \\ & j - i + 1 = w, \\ & s_i = t_i, \\ & s_{i+1} = t_{i+1}, \\ & \ldots \\ & s_j = t_j \} \end{aligned}$$

Alternately, for the lookahead pairs method, the profile P_{pair} is defined as:

$$\begin{aligned} P_{pair} = \{\langle s_i, s_j \rangle_l : s_i, s_j \in C, 2 \leq l \leq w \\ \exists p : \; 1 \leq p \leq \tau - l + 1, \\ t_p = s_i, \\ t_{p+l-1} = s_j \} \end{aligned}$$

To make these formal definitions more concrete, it is instructive to work through an example. Suppose we have chosen a window size of 4 ($w = 4$), and we observe a program producing this trace of system calls, with the earliest system call on the left:

position 3	position 2	position 1	current
			execve
		execve	brk
	execve	brk	open
execve	brk	open	fstat
brk	open	fstat	mmap
open	fstat	mmap	close
fstat	mmap	close	open
mmap	close	open	mmap
close	open	mmap	munmap

Table 4.1: A sample system call sequence profile.

position 3	position 2	position 1	current
			execve
		execve	brk
fstat	execve, mmap	brk, close	open
execve	brk	open	fstat
brk, mmap	open, close	fstat, open	mmap
open	fstat	mmap	close
close	open	mmap	munmap

Table 4.2: A sample lookahead pair profile, with the pairs represented implicitly. Note how there are multiple entries in the open and mmap rows.

execve, brk, open, fstat, mmap, close, open, mmap, munmap[1]

The first step in defining a profile for this trace is to record the contents of the overlapping length 4 windows. Let us number the four positions in this window as position 0, 1, 2, and 3, with position 0 being the most recent call, and 1 being the call immediately preceding it. For clarity, let us call position 0 the "current" position, since it refers to the most recently executed system call.

With these conventions, our trace produces the window contents shown in Table

[1]Actually, system calls are encoded as numbers between 0 and 255. The numbers corresponding to these calls are "11, 45, 5, 108, 90, 6, 5, 90, 91."

l	pairs (current, prev)
2	(brk, execve), (open, brk), (open, close), (fstat, open), (mmap, fstat), (mmap, open), (close, mmap), (munmap, mmap)
3	(open, execve), (open, mmap), (fstat, brk), (mmap, open), (mmap, close), (close, fstat), (munmap, open)
4	(open, fstat), (fstat, execve), (mmap, brk), (mmap, mmap), (close, open), (munmap, close)

Table 4.3: A sample lookahead pair profile, with the pairs represented explicitly.

4.1. For the sequence method, we would simply store the contents of this table. The incomplete sequences are also recorded, with an arbitrary "empty" value filling the unoccupied positions.

For the lookahead pair method, we compress the sequence representation by joining together lines with the same current value, as shown in Table 4.2. From this table, we can generate three sets of lookahead pairs: pairs for $l = 2$ (current and position 1), $l = 3$ (current and position 2), and $l = 4$ (current and position 3). Table 4.3 lists all of these lookahead pairs.

Given these profiles, the following sequence would be flagged as anomalous:

open, fstat, mmap, execve

The sequence method would signal that the execve was anomalous because this sequence is not listed in Table 4.1. The lookahead pair method would mark it as anomalous because the lookahead pairs (execve, mmap) ($l = 2$), (execve, fstat) ($l = 3$), and (execve, open) ($l = 4$) are all not present in Table 4.3.

This lookahead-pair information is what pH actually uses to monitor program behavior, with a window size $w = 9$. This data is stored in a 129K profile on disk, with one profile for every program that is run. They are loaded every time a new program is run and are saved when all processes using a given binary have exited.

For example, when a user types the *ls* command at a shell prompt, the shell forks a copy of itself. This child process then runs */bin/ls* through the use of the execve system call. Before this execve completes, pH loads the profile for */bin/ls* from disk. When the *ls* process terminates, pH saves its profile to disk and frees the profile's memory.

The next section analyzes the properties of the sequence and lookahead pair methods and provides some justification for why pH uses lookahead pairs.

4.3 Analysis

Having described two methods for representing normal program behavior at the system-call level, we need a set of criteria for choosing which one to use, and for choosing an appropriate window size for either method. Given the constraints of pH, we have a number of issues to consider:

- **Speed:** Above all else, we need a method that is fast enough to run in real-time without causing significant performance degradation. Thousands of system calls are made every second on a graphical workstation; accordingly, the analysis of each of these calls must be extremely efficient computationally.

- **Profile size:** We also need a method that generates small profiles of normal behavior. To some degree we can exchange size for speed; however, profiles should be significantly smaller than the typical executable size so as to minimize the load-time performance impact. Also, to protect against denial-of-service attacks there should be a reasonable upper bound on the size of a profile; otherwise, a randomly-behaving program could generate an arbitrarily large profile and exhaust system memory.

- **Generalization and Convergence:** The method should converge quickly, requiring a minimal amount of data to capture an approximation of normal program behavior. We cannot test for abnormal behavior until we have a model of normal behavior, so the faster the algorithm converges, the smaller the window of vulnerability will be. Note that a quickly converging method must generalize to previously unseen patterns of normal behavior; the greater the degree of generalization, the faster the speed of convergence, and the lower the level of false-positives.

- **Anomaly sensitivity:** The method must be able to clearly detect security-related anomalies. This requirement is inherently at odds with the previous requirement: the more normal behavior is generalized, the more likely it is that abnormal behavior is included in the normal profile.

The sections below compare the sequence and lookahead pair methods on the basis of these issues.

4.3.1 Speed

Both methods can be implemented efficiently so that lookups run in linear time proportional to the size of the window. The sequence method can be implemented using a hash table which stores each sequence, requiring a constant amount of time on average for the hash table lookup, and $O(w)$ comparisons to verify the presence of the sequence (assuming a low rate of collisions). If we can ensure a fixed number of buckets, insertion should take only slightly longer than a lookup; however, if the table fills up, then we may need to increase the size of the table, at a cost linear in the number of sequences stored. A tree implementation can provide better worse-case costs for insertion, but with somewhat higher lookup costs.

If lookahead pairs are stored in pre-allocated bit arrays, each pair may be looked up and modified in constant time. As there are $w - 1$ lookahead pairs in a w-size window, this means that both lookups and updates are guaranteed to run in $O(w)$ time, with a much simpler implementation than any sequence storage algorithms.

Due to factors such as cache fill penalties, the performance of these two methods may be closer in practice than the above analysis would indicate. Nevertheless, the inherent efficiency of the lookahead pair method makes it slightly preferable in terms of speed. The fact that this speed can be achieved with very simple code makes lookahead pairs even more attractive.

4.3.2 Profile Size

Our past research [43, 49] has indicated that a wide variety of programs normally produce only a few thousand unique sequences. Although both the sequence and lookahead pair methods can store this many sequences efficiently, these methods have have very different worst-case storage requirements. The lookahead pair table can be stored using a $c \times c$ bit array for each value of l. As a result, all possible lookahead pair patterns can be stored in $w - 1$ bit arrays with c^2 bits each, giving us a very reasonable upper bound of $O(wc^2)$. If we limit ourselves to $w = 9$ and round up c to 256 (the next largest power of 2), this means we can store any set of lookahead pairs (and hence the lookahead profile for any program) in a 256×256 byte array, or 64K bytes.

In contrast, there are c^w possible sequences of length w. If we let $w = 9$ and $w = 200$, this gives us 5.12×10^{20} possible sequences. If we represent each of these sequences by one bit, we may still need 58 million terabytes to store a profile. To be sure, in practice we would normally see many fewer sequences; a profile for a program making random system calls, though, could approach this limit. To prevent

extremely large profiles from causing a denial of service, we may limit the size of a sequence profile; inherent in sequence profiles, however, is the potential for exponentially large storage requirements.

4.3.3 Convergence

For $w > 2$, the lookahead pair method generalizes more than the sequence method, as the space of possible lookahead pair sets is $c^2(w-1)$ versus c^w possible sequences (as explained in the previous subsection). This increased generalization means that the lookahead method potentially can converge on a profile of normal behavior more quickly than the sequence method.

The actual rate of convergence depends on the distribution of patterns produced by a given program. Consider each position in our system-call window as a random variable. Let $\mathcal{P}_i$ be the random variable corresponding to the window position i, with $\mathcal{P}_0$ being the current system call. We can then define the random variables $\mathcal{L}$ for lookahead pairs and $\mathcal{S}$ for sequences as follows:

$$P(\mathcal{L}_l = \langle s_i, s_j \rangle_l) = P(\mathcal{P}_{l-1} = s_i, \mathcal{P}_0 = s_j)$$

$$P(\mathcal{S}_w = \langle s_i, s_{i+1}, \ldots, s_{i+w-1} \rangle) = P(\mathcal{P}_{w-1} = s_i, \mathcal{P}_{w-2} = s_{i+1}, \ldots, \mathcal{P}_0 = s_{i+w-1})$$

The contents of a profile can be thought of as an approximation of the joint distributions of these random variables as follows. For the lookahead method, the presence of a lookahead pair $\langle s_i, s_j \rangle_l$ in a program's profile implies that

$$P(\mathcal{L}_l = \langle s_i, s_j \rangle_l) > 0.$$

Similarly, the presence in a sequence profile of the sequence $\langle s_i, s_{i+1}, \ldots, s_{i+w-1} \rangle$ implies that

$$P(\mathcal{S}_w = \langle s_i, s_{i+1}, \ldots, s_{i+w-1} \rangle) > 0.$$

Lookahead profiles are therefore an approximation of the joint distribution of the current system call and a previous one, while sequence profiles approximate the joint distribution of all the system calls within the window. If we ignore edge effects, every call in a trace is eventually in every position of the window. Thus, the $\mathcal{P}_i$'s are dependent but identically distributed random variables.

We would like to start monitoring a program for anomalies after we have as good an approximation as possible of the probability distribution underlying a program's behavior. The difficulty of this task fundamentally depends on the distribution of $\mathcal{S}_w$ and $\mathcal{L}_l$. This distribution is a function of the number of unique sequences or lookahead pairs and the frequency of each. If a program can only produce a few types of sequences or lookahead pairs, then these will occur with high frequency and we will only need to train on a relatively short trace to see all normal patterns. Alternately, if a program is characterized by many distinct sequences and lookahead pairs, and if these patterns occur with roughly similar frequency, then on average we will have to observe a large amount of normal behavior before we can expect to see all of these patterns.

One quantification of this difficulty is the entropy of a random variable. Let

$$\mathbf{s}_w = \langle s_i, s_{i+1}, \ldots, s_{i+w-1} \rangle$$

$$\mathbf{l}_l = \langle s_i, s_{i+l-1} \rangle$$

Using the standard definition of entropy, we can define sequence and lookahead pair entropy as follows:

$$H(\mathcal{S}_w) = -\sum_{\mathbf{s}_w} P(\mathcal{S}_w = \mathbf{s}_w) \log_2 P(\mathcal{S}_w = \mathbf{s}_w)$$

$$H(\mathcal{L}_l) = -\sum_{\mathbf{l}_l} P(\mathcal{L}_l = \mathbf{l}_l) \log_2 P(\mathcal{L}_l = \mathbf{l}_l)$$

In Chapter 6, sequence and lookahead pair entropy are analyzed to see how they vary with window size and program type. Section 6.3 shows that programs with

larger code bases and greater functionality — the ones for which normal behavior is expected to be most varied — tend to have larger entropy values. It also shows that both methods are not overly sensitive to window size and that a window size of 9 works well with a wide variety of programs.

4.3.4 Anomaly Sensitivity

When evaluating the number of anomalies detected using different methods and window sizes, it is useful to consider what happens to the profile of a normal trace when a system call is added, substituted, or deleted.

When a call is added, up to w anomalous sequences are produced, one for each possible position of the new call. When the inserted system call is in the first position in the window, up to $w-1$ anomalous lookahead pairs are produced. The remaining $w-1$ sequences each maximally generate between $w-1$ and 1 anomalous lookahead pairs, with the number of possible anomalies decreasing as the inserted call moves further downstream. In total, these w anomalous sequences can generate up to $(w^2+w-4)/2$ anomalous lookahead pairs.

When a call is substituted, up to w anomalous sequences are produced. The first such sequence can generate up to $w-1$ anomalous lookahead pairs since the substituted system call is in the first position in the window; however, the rest can generate at most one anomalous lookahead pair, giving us a total of $2(w-1)$ possibly anomalous pairs.

A deleted call can produce up to $w-1$ anomalous sequences, one for each place the anomalous call could be inserted into the window. With lookahead pairs, the number of anomalous pairs depends on the position of the "hole" where the missing system call would be. When the hole is between positions 0 and 1, all $w-1$ lookahead pairs are potentially anomalous because the current call should have been the deleted

call. Moving the window over one position, the hole is between positions 1 and 2. The 0,1 pair is unperturbed by the hole, but the rest potentially are, giving us up to $w-2$ anomalous pairs for the entire window. The number of possible anomalous pairs continues to decrease as the hole moves across the window until it is between positions $w-2$ and $w-1$, where it can trigger at most 1 anomalous pair — the pair of positions 0 and $w-1$. In total, a deletion can generate up to $(w^2-w)/2$ anomalous pairs over $w-1$ sequences.

In all of these cases, if we consider one lookahead pair enough to flag a sequence as anomalous, both methods can produce the same maximum number of anomalous sequences. The sequence method, though, is more likely to produce a larger number of anomalies, especially in the case of a substitution: after the call moves from being the current system call, there is only one possibly anomalous lookahead pair. If just one other training sequence contained this pair, the lookahead method will consider the altered window (and thus the current system call) to be normal.

4.4 Summary

Because lookahead pairs can be updated and tested more quickly, generalize more, have modest worst-case storage requirements, and can be implemented easily, they have been used as the monitoring algorithm for pH. Sequences do have the advantage of potentially being more sensitive to anomalies; yet as Chapter 7 shows, the lookahead pair method is sensitive enough in practice.

Note that these criteria can also be used to compare the lookahead pair and sequence analysis methods with other algorithms. Hidden Markov models and rule-based induction algorithms (such as RIPPER) are competitive with these two methods on the basis of profile size, generalization, and anomaly sensitivity; they are much slower, however, in terms of training speed [116], making them inappropriate

for pH. Other methods such as finite-state machine machine induction algorithms [78] may be competitive with sequence and lookahead pair analysis in all four areas; more work needs to be done to see whether they would work well in a pH-like system.

Chapter 5

Implementing pH

Chapter 3 described the biological inspiration and high-level design of pH. This chapter explains how these ideas are implemented. The first section discusses why pH is implemented as a Linux kernel extension, and the next section gives an overview of the implementation. Sections 5.3, 5.4, and 5.5 explain how pH handles training and response. Section 5.6 presents the basic data structures used by pH, and Section 5.7 gives an overview of how pH's code integrates with the rest of the Linux kernel. Sections 5.8 and 5.9 summarize and discuss pH's runtime commands and parameters. The last section presents performance data which shows that pH has minimal overhead in practice.

5.1 Why a kernel implementation?

pH has three fundamental components: a mechanism that observes system calls, an analysis engine, and a response mechanism that slows down anomalous system calls. Because system calls are the basic interface between the kernel and userspace processes, these mechanisms could be implemented on either side of this interface;

with pH, though, all three components reside inside the kernel. To see why, it is worth reviewing how we have gathered system-call data in the past.

Before pH, most of our system-call traces [43, 49] were gathered using the *strace* program, a utility that uses the ptrace system call to monitor the system calls of another process. *strace* is extremely easy to use; unfortunately, *strace* would often cause security-critical programs like *sendmail* to crash, and when it worked, *strace* would slow down monitored programs by 50% or more.

These limitations led us to explore other options. Audit packages such as the SunOS Basic Security Module often record system-call data; however, such systems are slow, difficult to use, and they produce voluminous log files. Another approach was to insert the monitoring routines directly into the address space of the monitored program. To test its feasibility, I modified a SunOS C library (without source) to log all system calls made through library routines. The SunOS C library modifications were fast and efficient, and were used to observe *sendmail* on a production mail server at the MIT Artificial Intelligence Laboratory. By design, though, this monitor couldn't detect the system calls made by buffer overflow attacks, since the "shell code" of such attacks typically make system calls without using library routines.

Although the implementation of an in-kernel system call monitor was challenging at first, it quickly proved to be the most efficient, robust, and secure technique we had tried. In addition, kernel modification opened the door to response since a kernel-based mechanism can easily modify the behavior of system calls. By also placing the analysis engine in the kernel, all three mechanisms could interact without the use of voluminous log files. They could also now monitor every process on the system just as easily as monitoring one process. The drawback, though, was that all three pieces had to be fast, space-efficient, and robust. Because mistakes in kernel code often lead to system crashes, the robustness requirement was the most important one, and the hardest to achieve. It its current form, pH is robust and efficient enough to run

on production servers.

5.2 Implementation Overview

pH is implemented as a small patch for Linux 2.2 kernels (x86 architecture), consisting only of 3052 lines for pH version 0.18. This patch modifies the kernel so that pH is invoked at the start of every system call. When this happens, pH first adds the requested system call to a sequence that records recent calls for the current process. It then evaluates the current system call sequence, deciding whether it is anomalous or not. If it is anomalous, the current process is delayed by putting it to sleep, during which time other processes are scheduled to run. Once pH has finished, execution of the process's requested system call continues normally. This flow of control is illustrated in Figure 5.1.

To determine whether a program is behaving abnormally, pH maintains a profile containing two datasets for every executable, a "training" dataset and a "testing" dataset. The relationship between these two datasets is shown in Figure 5.2. These datasets represent lookahead pairs using bit arrays. Once pH has been activated, pH continuously adds the lookahead pairs associated with a process's current system call sequence to its training dataset (i.e. to the training dataset of the profile associated with the process's current executable).

Once no new pairs have been added to a training dataset for a sufficient period of time, or at a user's request, this training dataset is copied to the profile's testing dataset. This testing dataset is then used by pH to classify sequences as being normal or abnormal, based on whether the lookahead pairs for a given sequence are present in the testing dataset. Note that the training dataset is updated even when a profile has a valid testing dataset.

When a process loads a new executable via the execve system call, pH loads from disk the executable's profile. If a process is behaving sufficiently abnormally, though, pH delays for two days any execve call made by that process, effectively disabling that process's ability to load new programs until a user has an opportunity to evaluate the process's behavior. As Chapter 7 explains, this mechanism is necessary to defeat certain kinds of attacks.

Users interact with pH either by using two command-line utilities, *pH-ps* and *pH-admin*, or through the graphical *pHmon utility* (see Figure5.3). Both *pHmon* and *pH-admin* allow users to change parameters and modify the state of specific profiles and processes. pH provides four basic commands that operate on processes, but which affect both the process and its associated profile:

- **Reset:** Erase the process's profile (training & testing datasets).
- **Normalize:** Start normal monitoring.
- **Sensitize:** Forget recently learned program behavior.
- **Tolerize:** Accept recent program behavior.

The precise function of these commands is explained in Section 5.8. The names of the last two actions are inspired by analogous processes in the human immune system, and in effect allow a user to manually classify recent program behavior. By "sensitizing" a profile, we are telling pH that recent program behavior was abnormal and should not be added to the program's profile. On the other hand, "tolerize" tells pH that it incorrectly classified a program's behavior as abnormal, and therefore pH should cancel normal monitoring. A tolerized profile may eventually become normal again, but only after the training profile has re-stabilized.

pH is distributed under the GNU General Public License (GPL), and can be downloaded from http://www.cs.unm.edu/∼soma/pH/.

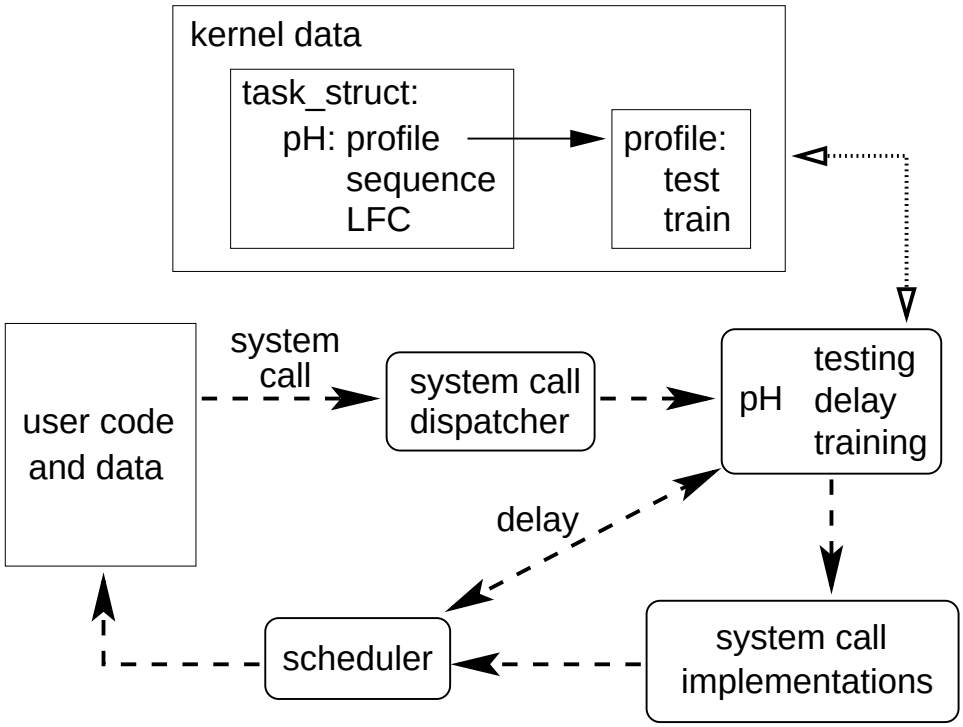

Figure 5.1: Basic flow of control and data in a pH-modified Linux kernel.

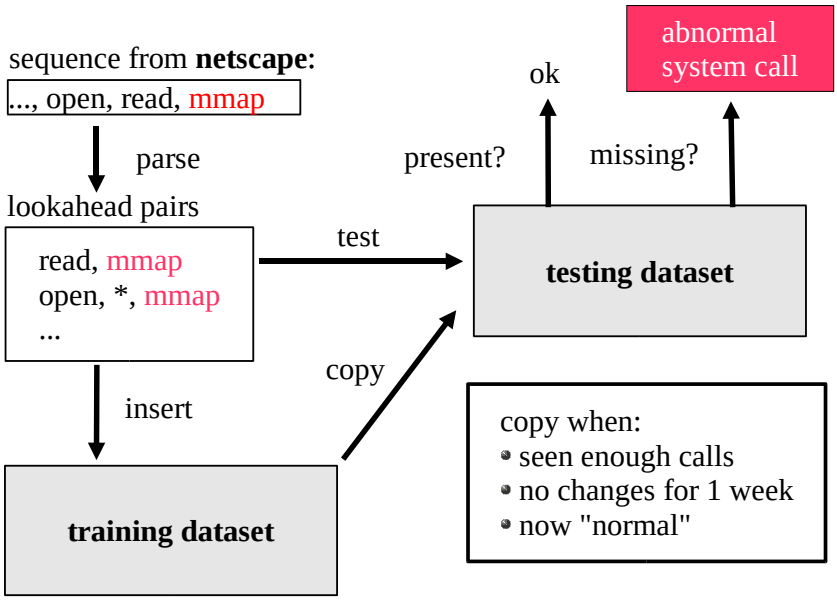

Figure 5.2: Relationship between the training and testing datasets in the profile of *netscape*.

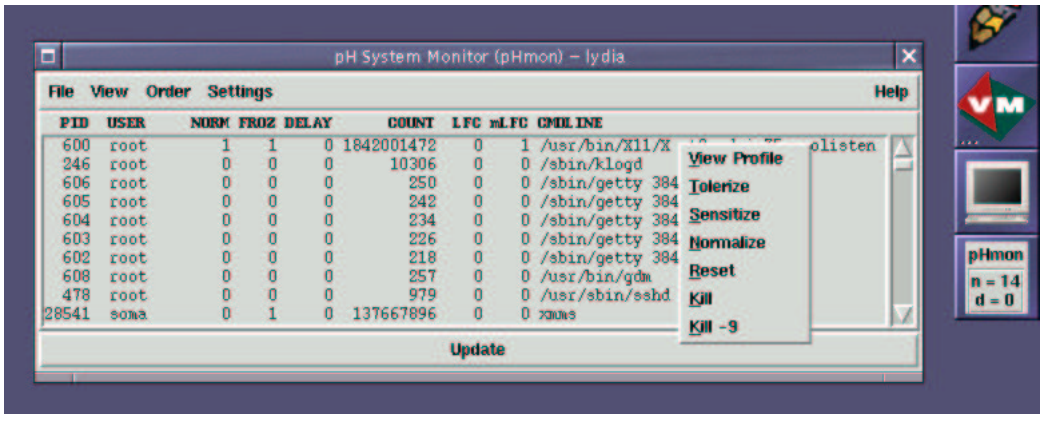

Figure 5.3: Screenshot of the *pHmon* pH monitoring utility, after a process has been selected. Notice the actions available in the pop-up menu.

5.3 Classifying Normal Profiles

5.3.1 Requirements

As previously mentioned, pH maintains two sets of data in each program's profile: a training dataset and a testing dataset. If an observed sequence is not present in the training dataset, it is added. If it is not present in the testing dataset, an anomaly is generated. Initially, the testing dataset is empty, and the profile is considered "not normal," i.e. it does not represent the normal behavior of the program. Once there is a valid testing dataset, the profile is said to be "normal." So, before pH can start detecting anomalies, it must have a valid testing dataset. Because the testing dataset is never updated directly, this dataset must be copied from the training dataset. What we need, then, is a set of conditions under which this copying takes place.

Rather than having a person manually trigger copying after having decided that enough behavior has been observed, we want pH itself to detect when the training dataset has stabilized and perform the copy autonomously. Because pH resides in kernel space, this analysis must be computationally inexpensive; however, it must also be accurate, otherwise pH will detect too many false positives or it will miss real attacks.

To detect this stabilization, we must observe (potentially in an extremely simplified form) the distribution of novel system call sequences, both in terms of novel sequences per system call and novel sequences per unit time. If we just consider the former, the behavior of a program making frequent system calls may appear to stabilize after only a few seconds and then may proceed to generate false positives only a few minutes after being invoked. If we consider just the latter, then the profile for a program that is run only occasionally will become normal prematurely, again

generating unwanted false positives.

pH's normal-classification mechanisms are based on simple, efficient heuristics. Even though these heuristics were created on an ad-hoc basis, as Chapter 6 shows, they work well in practice.

5.3.2 Profile Heuristics

pH employs a two-stage algorithm to decide when a given profile is normal. The first stage involves a simple heuristic, explained below, which decides whether the profile has stabilized based on the number of calls made since the last change to the profile. If this heuristic is true, then the profile is marked as *frozen*, and the time of this event is recorded. If a program produces a novel sequence while the profile is frozen, the profile is then "thawed," and the sequence is added to the profile. The second stage simply checks to see whether a profile has been frozen for a certain amount of time. If it has, then the profile is marked as normal.

The first stage's heuristics depend on two observed values, *train_count* and *last_mod_count*. *train_count* is the number of system calls seen during training; usually, it is the number of calls executed by all processes running the profile's executable. If a program has been invoked twice, with each invocation making 500 system calls, the *train_count* for the corresponding profile will be 1000. *last_mod_count* is the number of calls that have been seen since the last change to the profile. Every time a sequence is added to a profile, its *last_mod_count* is set to 0; otherwise, this count is incremented on every system call made by processes running a profile's program. The variable *normal_count* is also used in the heuristic calculations, but it is simply the value of *train_count* minus *last_mod_count*. Conceptually, *normal_count* is the number of calls seen between the first added sequence and the last added sequence. Figure 5.4 shows the relationship between these three values.

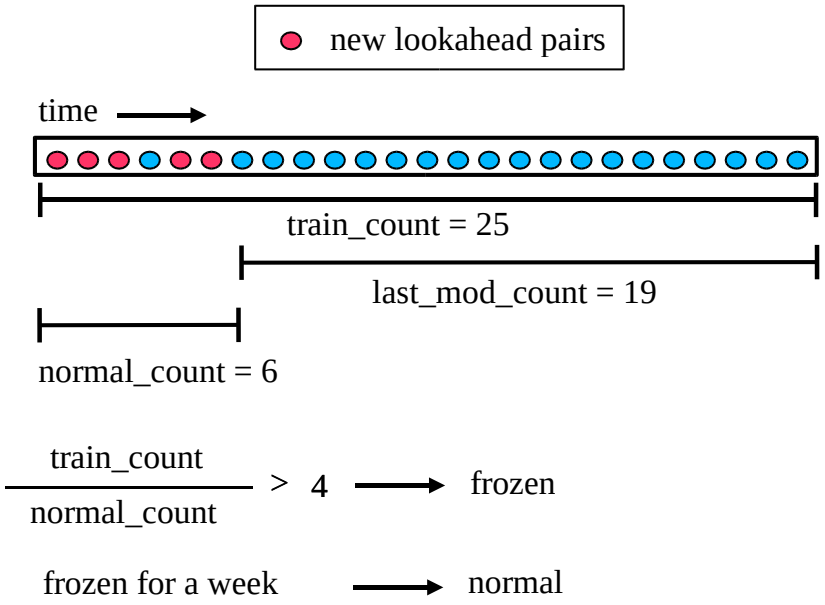

Figure 5.4: pH's normal profile heuristics.

A profile is frozen when two conditions are met:

$$normal\_count > 0$$

and

$$\frac{train\_count}{normal\_count} > \frac{normal\_factor}{normal\_factor\_den}$$

Where $normal\_factor$ is a runtime parameter and $normal\_factor\_den$ is a compile-time parameter. Note that in practice, this equation is cross-multiplied to remove the need for division. By default, $normal\_factor = 128$ and $normal\_factor\_den$ is 32, giving us a ratio of 4.

To freeze a profile, two actions must be performed. First, a flag in the profile

called $frozen$ is changed from 0 to 1. Next, pH records a time which is the current time plus *normal_wait*. By default, this is 604800 seconds, or one week. If this time passes and the profile is still frozen, the profile is made normal.

There are four mechanical steps pH takes to make a profile normal. First, the training lookahead pair dataset is copied to testing. Then, the *train_count* and *last_mod_count* variables are set to 0. And finally, the *normal* flag is set to 1, while the $frozen$ flag is set to 0.

Note that by setting *train_count* and *last_mod_count* to 0, *normal_count* also becomes 0, and in fact will stay 0 as long as no new sequences are encountered.

If a profile is frozen, and the program executes a previously unseen sequence of system calls, the profile is quietly thawed by setting the $frozen$ flag to 0.

5.4 Delaying System Calls

As explained in Section 3.2.6, pH responds to anomalous system calls by delaying them. On the assumption that security violations will tend to produce more anomalous system calls than other, more benign variations in program behavior, pH performs delays in proportion to the number of recently anomalous system calls.

To implement proportional delays, pH records recent anomalous system calls in a fixed-size circular byte array which we refer to as a *locality frame*. More precisely, let n be the size of our locality frame, and let A_i be the i-th entry of the locality frame array, with $0 \leq i < n$ and $A_i \in \{0, 1\}$. Also, let t_0 be the first system call executed by the current process, t_k be the current system call for that process, and let t_{k-1} be the previous system call. Initially, the entries of A are initialized to 0. Then, as the process runs and executes successive t_k system calls, A is modified such that $A_{k \bmod n} = 1$ if t_k is anomalous, and 0 otherwise. As the process runs, A contains the

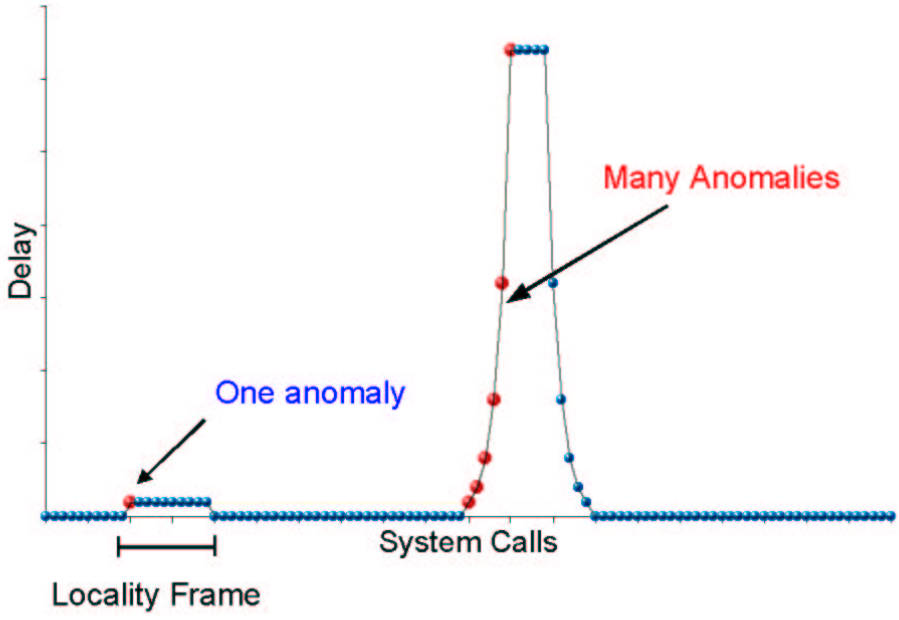

Figure 5.5: A schematic graph showing how a process is delayed in response to different-sized clusters of anomalies.

record of how many of the past n system calls were anomalous. We call the total of recent anomalies, $\sum A_i$, the locality frame count (LFC)[1].

If the LFC for a process is greater than zero, pH delays each system call for $delay\_factor \times 2^{LFC}$, even if the current system call is not anomalous. This continued response ensures a relatively smooth increase and decrease in the response to a cluster of anomalies, as shown in Figure 5.5.

[1]A somewhat different approach was taken in Hofmeyr [49], where the measure of anomalous behavior was based on Hamming distances between unknown sequences and their closest match in the normal database. Although this method provides a direct measure of the distance between a sequence and a normal profile, it requires significantly more computation to calculate, and so is less suitable for an online system.

As explained previously, pH maintains profiles on a per-executable basis. A problem with this approach is that an execve can allow an anomalously behaving program to escape pH's monitoring. For example, a buffer overflow attack may cause the vulnerable program to perform only one anomalous system call, an execve. After this call, pH will then look for anomalies relative to the new binary. The newly exec'd program may be behaving completely normally while in fact the entire invocation of the program is anomalous[2].

To counter this problem, pH treats the execve system call as a special case. pH keeps track of the maximum LFC value seen for a process. If this value exceeds a certain threshold, pH interferes with execve calls. In earlier versions of pH (pH-0.17 and earlier), if the maximum LFC value for a process equals or exceeded the *abort_execve* threshold, any execve calls made by that process would automatically fail. While this mechanism did solve the problem, it also caused false-positives to change the semantics of a program.

To avoid this side effect, more recent pH versions use the *suspend_execve* threshold. If the maximum LFC value for a process equals or exceeds *suspend_execve*, then the program is delayed for *susp_exec_time* seconds. Unlike *abort_execve*, erroneous *suspend_execve* responses are generally easy to recover from: a user cancels the delay, and the program continues from where it left off. An attacker, though, is stopped in his or her tracks for *susp_exec_time* seconds, which by default is two days. Even if the attacker is patient, an administrator thus has plenty of time to decide whether an attack was genuine and could kill the offending process before the attack succeeded.

[2]In much of our past work [43, 49], execve calls did not cause a new profile to be loaded. Thus, if *sendmail* loaded *bash* using execve, the system calls of *bash* were evaluated relative to *sendmail*'s behavior profile. As a result, it was extremely easy to detect buffer overflow attacks, because *bash*'s behavior is very different from *sendmail*'s. This strategy, though, does not work if we want to monitor every program on a system automatically, especially if we wish to use more than one profile.

5.5 Tolerization and Sensitization

As explained in Section 5.2, users can manually classify recent program behavior through the "tolerize" and "sensitize" commands. pH can also perform both actions on its own. Automatic tolerization and sensitization are controlled through the *anomaly_limit* and *tolerize_limit* parameters, respectively.

The *anomaly_limit* is used by pH to ensure that pH eventually stops responding to minor anomalies. pH keeps a count of the total number of anomalies generated by each profile. If this value ever exceeds *anomaly_limit* (which by default is 30), then the profile is tolerized by setting its normal flag to 0, invalidating its testing dataset. The process is also tolerized by resetting its locality frame and canceling any pending delays. For pH to reclassify the profile as normal, the training dataset has to again meet the criteria outlined in Section 5.3.

In contrast, the *tolerize_limit* threshold is used to prevents pH from eventually learning an attack for which it has mounted a significant response. If the LFC for a process ever exceeds *tolerize_limit* (which by default is 12), the corresponding profile is sensitized by resetting the profile's training dataset. This action causes all previously learned training lookahead pairs to be erased. The profile's testing dataset is preserved, however, allowing pH to continue to respond to anomalous program behavior. In addition, the anomaly count for the profile is set to 0. This second mechanism prevents the *anomaly_limit* threshold from being achieved while pH is mounting a significant response to a process.

To see how these thresholds interact in practice, consider a profile that has just been classified as normal. If subsequent invocations of the associated program generate 20 anomalies each, but achieve a maximum LFC of only 10, these program runs will never exceed the *tolerize_limit* threshold, and thus the training dataset for the profile will not be reset. On the program's second invocation, however, the

anomaly count for the corresponding profile would exceed *anomaly_limit*, causing normal monitoring for that profile to be canceled. Because the profile is no longer "normal," pH now would not delay subsequent program runs.

If the same program's 20 anomalies had been closer together and had generated a maximum LFC of 20, then the role of the two thresholds would be reversed: *tolerize_limit* would be exceeded, the training dataset would be reset, and the profile's anomaly count would be set to 0. The *anomaly_limit* threshold would never be exceeded, and without user intervention, every future invocation of the program would be delayed by pH.

Note that these two thresholds are invoked in practice. Chapter 6 presents some results on the frequency of automatic tolerization, and Chapter 7 gives some examples of pH automatically sensitizing a profile in response to an attack. These results show that although these two thresholds are crude and somewhat arbitrary, they do allow pH to respond appropriately both to the concentrated anomalies of a security violation and to the more distributed anomalies of many kinds of false positives.

5.6 Data Structures

In the Linux kernel, processes and kernel-level threads are both implemented in terms of *tasks*. A task is a kernel-schedulable thread of control, and is represented internally by a *task_struct* structure. If a task has its own virtual address space, it is a complete, single-threaded process. If a task shares its address space with another task, it is one thread of a multi-threaded process. Like the Linux kernel, pH does not distinguish between processes and threads and instead operates on running programs as tasks.

pH has two fundamental data structures: `pH_task_state` and `pH_profile` (Fig-

ures 5.6 and 5.7). A `pH_profile` is maintained for every executable currently running and a `pH_task_state` is kept for every task. Together, these hold the data needed for pH to monitor programs and respond to anomalies.

Each `task_struct` structure contains information such as the user ID, the process (task) ID, the parent and children processes, which program the process is executing, and virtual memory allocations. As shown in Figure 5.6, pH adds one more field to this structure, `pH_state`, for a `pH_task_state` structure. A `pH_task_state` holds information on the task's locality frame, system call sequence, delay, and a pointer to the profile for the process.

Multiple tasks may run the same executable, e.g., there may be several running instances of *bash* or *emacs*. pH maintains one `pH_profile` structure per executable, and these are shared amongst the tasks running that executable: thus, the `pH_task_struct` for two processes running *emacs* will both point to the same `pH_profile`. The `pH_profile` structure contains information on whether this profile is normal and holds information on the training and testing lookahead pairs for this executable. Figure 5.7 shows a simplified version of the C `pH_profile` structure definition. In the actual pH code, in-kernel storage for the lookahead pair datasets is dynamically allocated as 4K pages. The on-disk profile format, though, uses static arrays; therefore, while profiles require 129K on disk, they require as little as 8K in memory (one page for the main profile, and one for a small training dataset).

```
#define PH_LOCALITY_WIN 128
#define PH_MAX_SEQLEN 9

typedef struct pH_locality {
        unsigned char win[PH_LOCALITY_WIN];
        int first, lfc, max_lfc;
} pH_locality;

typedef struct pH_seq {
        int last, length;
        u8 data[PH_MAX_SEQLEN];
} pH_seq;

typedef struct pH_task_state {
        pH_locality lf;
        pH_seq seq;
        int delay;
        unsigned long count;
        pH_profile *profile;
} pH_task_state;

struct task_struct {
        ...
pid_t pid;
...
pH_task_state pH_state;
};
```

Figure 5.6: Simplified `pH_task_state` definitions.

```
#define PH_NUM_SYSCALLS 256   /* size of array */

typedef unsigned char pH_seqflags;

typedef struct pH_profile_data {
        unsigned long last_mod_count;
        unsigned long train_count;
        pH_seqflags entry[PH_NUM_SYSCALLS][PH_NUM_SYSCALLS];
} pH_profile_data;

typedef struct pH_profile pH_profile;

struct pH_profile {
        int normal;           /* is test profile normal? */
        int frozen;           /* is train profile frozen */
        time_t normal_time;   /* when will frozen become normal? */
        int window_size;
        unsigned long count;  /* # calls seen by this profile */
        int anomalies;
        pH_profile_data train, test;
        pH_profile *next;
};
```

Figure 5.7: Simplified `pH_profile` definitions.

5.7 Kernel Integration

In order for pH to monitor system call behavior, delay anomalous system calls, and communicate with users, pH must interact with the rest of the Linux kernel in several ways. This section describes the key pH functions, and explains their purpose and how they integrate with the rest of the kernel.

The `pH_process_syscall()` function is called by the system call dispatcher just before running a requested system call and is the primary connection between pH and the rest of the kernel. This routine adds the requested call to the task's sequence (stored in `pH_state`), and then, if necessary, adds the sequence to the profile's training dataset. Next, it updates the task's locality frame, adding a 1 if the task's profile is normal and the current sequence is anomalous, or a 0 otherwise. It then decides whether the training profile should be frozen and whether a frozen training profile should be copied to testing, making the profile normal. Finally, if the locality frame count is greater than 0, the task is put to sleep for 0.01×2^{LFC} seconds. After `pH_process_syscall()` completes, control returns to the system call dispatcher, which then proceeds to run the task's requested system call.

`pH_do_suspend_execve()` is run near the beginning of every execve system call. If the current task's LFC is greater than or equal to the *suspend_execve* threshold, the task is delayed for *susp_exec_time* seconds. After this function completes, the execve system call is resumed.

The `pH_execve()` function is invoked by the execve system call just before it returns. This function first finds the correct profile for the program that has just been loaded. pH checks a linked list of loaded profiles to see whether the program's profile is already in memory. If it is not, pH attempts to load the program's profile from disk. Profiles are stored in a directory tree that mirrors the rest of the filesystem, but rooted in `/var/lib/pH/profiles`. (This path can only be

changed at compile-time.) For example, if a task runs `/bin/ls`, pH loads the profile `/var/lib/pH/profiles/bin/ls`. If a profile for a given executable does not exist, pH creates a new profile.

After finding the appropriate profile, `pH_execve()` initializes the `pH_state` field of the task's `task_struct`: it changes the `pH_state.profile` pointer to refer to the new profile and re-initializes `seq`, the current system-call sequence. Note that `pH_state.lf`, the current locality frame, is preserved, allowing pH to continue delaying anomalously behaving tasks. After this function completes, the task is monitored, whether it previously was or not.

`pH_fork()` is invoked just before the *fork* system call is completed. It copies the `pH_task_state` from parent to child (including the task's system call sequence and locality frame), ensuring that the child task will be monitored in the same way as the parent.

When a task exits via the *exit* system call, the `pH_exit()` function de-allocates the storage used for that task and writes any freed profiles to disk.

There are two mechanisms for interacting with pH: the pH system call, and the `/proc` virtual filesystem. The `sys_pH()` function implements the pH system call, which allows the superuser to change the parameters of pH and modify the status of monitored tasks.

The `/proc` virtual filesystem allows users to view the state of pH. The function `pH_proc_status()` provides the contents of `/proc/pH`, which shows information such as parameter values and the total number of system calls pH has processed. In addition, the monitoring status of every process is available in `/proc/<PID>/pH`. This file is generated by `get_pH_taskinfo()`, and it reports the values of the process's `pH_state`. These status files are used by monitoring programs to determine when pH is delaying anomalously behaving processes.

Command	Argument	Description
on, off	none	turn system-call monitoring on and off
status	none	write pH's status to kernel log
write-profiles	none	write all profiles to disk
log-syscalls	0/1 (off/on)	logging of all system calls
log-sequences	0/1 (off/on)	logging of system call sequences
tolerize	process ID	cancel normal (invalidate testing)
sensitize	process ID	reset training
normalize	process ID	copy training to testing
reset	process ID	reset training and testing

Table 5.1: The commands of pH. In addition, there are commands for changing all runtime parameters.

5.8 Interacting with pH

Although pH's basic functionality resides inside the Linux kernel, the `/proc` virtual filesystem and the pH system call allow userspace programs to interact with pH. The pH distribution includes three programs, *pH-ps*, *pH-admin*, and *pHmon*, that use these interfaces to monitor and change pH's behavior.

pH-ps is a command-line program that scans through the `/proc/<PID>/pH` files and summarizes their contents in a *ps*-like format. *pH-admin* is a command-line utility that exposes the functionality of the pH system call. For convenience, it is normally configured to run as the superuser; otherwise, only the superuser can change pH's behavior.

pHmon is a graphical utility that combines the functionality of *pH-ps* and *pH-admin*. A screenshot of *pHmon* is shown in Figure 5.3. With the WindowMaker X11 window manager, *pHmon* maintains a dockable application icon that summarizes the pH status of programs on the system. This icon shows how many processes currently have normal profiles (the n value), and how many processes are being delayed (the

d value). If one or more processes are ever delayed, the colors of this icon change to be red letters on a black background, providing a clear visual indicator that one or more processes are behaving unusually. Clicking on this icon brings up a process list window. The view in this window may be sorted and pruned using the "Order" and "View" menus. The state of any process can be changed by clicking on it and selecting the desired command from the pop-up menu. (*pHmon* runs *pH-admin* to actually execute the command.) The "Settings" menu lists the current values of pH's parameters. Selecting a parameter from this menu brings up a dialog window for changing its value.

The commands available through *pHmon* and *pH-admin* are summarized in Table 5.1. The "on" command instructs pH to begin monitoring of system calls. After this command is executed, every execve system call causes pH to load a profile from disk for the requested executable. If a profile does not exist for that program, a new one is created. pH then proceeds to monitor subsequent system calls of that process. The "off" command causes pH to write all profiles to disk and cease monitoring system calls.

The "status" command tells pH to log pH's current status to the kernel log. This status message is very similar to the contents of `/proc/pH` and contains all of pH's parameter settings and the number of system calls monitored by pH.

The "write-profiles" command tells pH to write all currently loaded profiles to disk. Normally, pH only updates the on-disk profile when every process running an executable exits. Some programs, such as web servers, run continuously, and so their on-disk profile is rarely updated. This command is used to ensure that the on-disk profiles are kept up-to-date.

The "log-syscalls" command tells pH to log every system call executed to a compile-time specified binary file, by default `/var/lib/pH/all_logfile`. This file

records the call number, process ID, the time of the call, and the current system call count of every system call. For forks, the child process ID is logged, and for execve system calls, the filename of the requested program is recorded. Together, this information can be used to retrace and analyze the actions of pH. One problem with this command is that even though each system call takes only 17 bytes to record, this file grows *very* quickly. Further, this file cannot be read while it is in use without creating a feedback loop: a read of the file is logged in the file, causing it to grow, causing a read to be logged, and so on. This situation can be avoided by using a technique borrowed from *syslog*: `/var/lib/pH/all_logfile` is moved to a new filename, and then the "log syscalls 1" is made. pH then closes the old logfile and opens a new `all_logfile`, all without losing any system call data. The *pH_print_syscalls* program in the pH distribution parses `all_logfile` logfiles into a human-readable format.

"Log-sequences" instructs pH to log each system-call sequence that is not represented in a program's training lookahead pair profile. This binary logfile has the same name is the program's profile, except with a ".seq" appended to it. Thus, the sequences for `/bin/ls` are stored in `/var/lib/pH/profiles/bin/ls.seq`. The sequence, the time of its addition, the profile's current system call count, and the process ID are stored in this file. The *pH_print_sequences* program in the pH distribution parses these logfiles into a human-readable format.

The "tolerize" and "sensitize" commands work as outlined in Section 5.5. "Normalize" manually marks a process's profile as normal, following the procedure outlined in Section 5.3. "Reset" resets the state of a process's `pH_state` and the associated `pH_profile`, making it appear that monitoring had just begun for the selected process. Note that all four of these commands operate on a process and its associated profile (task). Because multiple processes can share the same profile, these commands can indirectly affect other processes; however, these commands only change

Parameter	Default	Description
default_looklen	9	lookahead pair window size
locality_win	128	locality frame window size
normal_factor_den	32	denominator for normal_factor
loglevel	3	logging verbosity level (0=none)
normal_factor	128	ratio for freezing heuristics
normal_wait	604800	seconds for frozen to normal (7 days)
delay_factor	1	delay scaling factor (0=no delays)
suspend_execve	10	process anomalies before execve delay
suspend_exeve_time	172800	execve delay time seconds (2 days)
anomaly_limit	30	profile anomalies before auto-tolerization
tolerize_limit	12	process anomalies before train reset

Table 5.2: The parameters of pH. Parameters in boldface can only be set at compile time; the rest can be set at runtime.

the `pH_state` of one task. For example, consider four processes that are running *bash*. If all four are being delayed, tolerizing one will prevent the other three from generating any new anomalies. Tolerization, however, does not erase the contents of the other locality frames; therefore, the other three processes to continue to be delayed, even though they no longer have a normal profile. All four processes must be manually tolerized if they are to run at full speed.

5.9 Parameters

The previous sections have referred to several parameters. Table 5.2 summarizes these parameters and their default values. The first three can only be set at compile time, while the rest can be modified at runtime via the pH system call. To better understand the rationale for these values, it is useful to explore how the values of these parameters interact and affect the behavior of pH. The following parts discuss related groupings of pH's parameters.

5.9.1 System Call Window Sizes

First, consider the two compiled-in window size parameters. The $default\_looklen$ parameter specifies the size of the window used to record recent system calls. As explained in Chapter 4, larger window sizes can increase anomaly sensitivity, but at the cost of longer training times and larger lookahead pair storage requirements. The first part of Chapter 6 presents results that show that although there isn't an optimal window size, windows of sizes ranging from 6 to 15 or so work well on average. pH uses a default window size of 9 because this is the largest window that can be represented using the eight bits of an unsigned char. Larger window sizes would require `pH_seqflags` in Figure 5.7 be changed to a larger type; however, doing so would at least double a profile's size.

The size of the locality frame, *locality_win*, determines the window in which pH can correlate multiple anomalies. In earlier work [44], we arbitrarily used a frame size of 20. Experiments performed with pH and an *ssh* backdoor [106], though, revealed that anomalies for one attack could be separated by 35 system calls. If we assume that other attacks might have even more widely separated anomalies, it made sense to choose a significantly larger value for *locality_win*. Since computers tend to prefer memory chunks in sizes that are a power of two, I chose the default value of 128. Although this value is arbitrary, it works well in practice.

5.9.2 Logging

loglevel controls the type of logging messages sent to klogd, the kernel logging daemon. A level of 0 means don't generate any messages. A level of 1 means that pH should only log errors. $loglevel = 2$ tells pH to log changes in state, such as parameter changes and the starting of normal monitoring. $loglevel = 3$ causes pH to generate a message every time it detects an anomaly or delays a process. Finally,

$loglevel = 4$ produces messages every time pH reads, writes, or creates a profile. Messages for each level include those of the previous level. Thus, a loglevel of 4 or greater gives maximum verbosity. By default, *loglevel* is set to 3, causing pH to produce all log messages except those involving profile I/O.

5.9.3 Classifying Normal Profiles

$normal\_factor$ is used in the normal classification heuristics explained Section 5.3. Rather than using the form outlined there, pH instead uses this form of the test to avoid division operations:

$$(train\_count)(normal\_factor\_den) > (normal\_count)(normal\_factor)$$

$normal\_factor\_den$ was added to allow $normal\_factor$ to represent fractional values without using floating-point numbers. This change was necessary because floating point variables cannot be used inside the Linux kernel. $normal\_factor\_den$ should be a power of 2 so the compiler can use a shift instead of a multiply instruction; other than this constraint, it merely needs to be large enough to provide an adequate fractional range for $normal\_factor$. Note that $train\_count \geq normal\_count$, always. Therefore, to keep this ratio from always being true, $normal\_factor$ should never be less than $normal\_factor\_den$ (32).

The default $normal\_factor$ of 128 seems to work well in practice. This value gives us a $train\_count/normal\_count$ of 4, which means that a program has made 3/4 of its total system calls without producing any new lookahead pairs. Values such as 64 and 48 tend to make pH a bit too aggressive in its classifications of normal profiles, causing rarely used programs to be frozen prematurely. Values much greater than 128 cause pH to be too conservative, resulting in fewer normal profiles.

When choosing a value for $normal\_factor$, we need to consider it in the context of $normal\_wait$. The default value of a week in seconds for $normal\_wait$ means that

programs which pass the $normal\_factor$ ratio test must also not change for a week. This value is longer than necessary for frequently used programs; however, it is just long enough to capture weekly *cron* jobs and other low-frequency activities.

One thing that $normal\_wait$ does not specify is how much a program must be used during its one week waiting time. If a program is used only occasionally, it may sit on disk for a week without being used. If its profile was frozen before a quiet week, it is possible it will not make any system calls before its profile is classified as normal. Thus, if there are many programs that are used infrequently with respect to $normal\_wait$ seconds, then $normal\_factor$ needs to be large, potentially 128 or greater. Alternately, if there are few infrequently used programs, $normal\_factor$ can be 64 or less, causing more profiles to be frozen. Of course, a better solution would be for pH to also consider how frequently a program is run before classifying it as normal; currently, though, pH does not have this capability.

One implementation detail to note: because pH uses 32-bit unsigned integers for its calculations, programs that make more than approximately 135 million system calls will cause one or both sides of this equation to wrap around to 0. This apparent flaw is actually an advantage in practice because it can cause pH to freeze profiles that otherwise wouldn't be frozen. If these active profiles also pass the $normal\_wait$, the resulting normal is relatively stable; further, since the program makes a large number of system calls, it is probably also an important program.

5.9.4 Automated Response

The last five parameters in Table 5.2 control different aspects of pH's automated response. The value of the first of these, $delay\_factor$, determines the intensity of pH's reaction to anomalous system calls. If $delay\_factor$ is 0, then no delays are

performed; otherwise, all system calls are delayed for

$$delay\_factor \times 0.01 \times 2^{LFC}$$

seconds. Given the default *locality_win* of 128 and $delay\_factor$ of 1, a process that produces one anomalous system call will be delayed in total for 1.28 seconds, provided it makes 127 system calls after the anomaly. Higher values for $delay\_factor$ can thus change this 1.28 seconds to be an arbitrarily large period of time. I have found that if delays are going to stop an attack, it will be stopped with $delay\_factor$ set to 1. Higher values make pH react more severely to anomalies, but this only tends to make false positives more problematic.

The values of *suspend_execve* and *tolerize_limit* are closely related: the first says the LFC value at which pH should delay execve requests, while the second tells the LFC which should cause a profile's training dataset to be erased. I see the *tolerize_limit* as the LFC value which implies dangerous program behavior, while *suspend_execve* denotes the LFC value where one should suspect the behavior of a program. If we assume that most program behavior is acceptable, then *tolerize_limit* should be high enough so that it is rarely invoked. On the other hand, *suspend_execve* should be set to the value where an execve should require human intervention.

The default value of 12 for *tolerize_limit* is slightly high, but is low enough that attacks often cause training data to be discarded. For more aggressive protection, a value of 5 or 6 would be better.

On the other hand, the value of 10 for *suspend_execve* is very high and effectively prevents execve's from ever being delayed. As Chapter 7 shows, it is necessary to set *suspend_execve* to 1 to defend against buffer overflow attacks. As discussed in Section 6.9, though, such a low setting requires that someone monitor pH's behavior on a regular basis.

anomaly_limit should be small enough to allow chronic problems to be tolerized eventually while being large enough to minimize the likelihood that an attacker can train pH by running an attack multiple times. *anomaly_limit* must be larger than *tolerize_limit*, otherwise normal monitoring will be canceled before *tolerize_limit* can be reached. Although this value has not been optimized, *anomaly_limit*'s default value of 30 seems to work well in most situations.

5.10 Performance

In order to be usable on production systems, pH needed to be fast and efficient enough so that users would not be inconvenienced by its overhead. Although there are no firm guidelines, one rule of thumb is that users are not happy with security mechanisms that slow down a system by more than 10%, and that to be truly unobtrusive, the slowdown should be 5% or less. Because pH significantly changes system-call dispatches, pH adds significant overhead to each system call; this overhead, though, does not cause regular applications to be slowed down by more than 5%.

To determine the performance impact of pH, I ran the HBench-OS 1.0 [15] low-level benchmark suite on lydia, a Hewlett-Packard Pavilion 8260 (266 MHz Pentium II, 288M SDRAM, Maxtor 91020 10G Ultra-DMA IDE hard disk) running a pre-release version of Debian/GNU Linux 3.0 (woody) Linux distribution. Tests were run for ten iterations on a system running in single user mode. In Tables 5.3 and 5.4, "Standard" refers to a Linux 2.2.19 kernel patched with the Linux 2.4 IDE back-port (05042001) and reiserfs 3.5.33. "pH" refers to the same kernel, but patched with pH-0.18. The pH kernel had monitoring enabled for all processes and was logging status messages. pH's delay mechanisms, however, were disabled.

Tables 5.3 and 5.4 show that pH does add significantly to system call overhead. Table 5.3 indicates that pH adds 1.9 μs to the execution time of simple system calls

System Call	Standard (μs)	pH (μs)	% Increase
getpid	1.162 (0.0001)	3.079 (0.0010)	165.0
getrusage	2.346 (0.0000)	4.247 (0.0010)	81.0
gettimeofday	1.672 (0.0005)	3.574 (0.0042)	113.8
sigaction	2.545 (0.0001)	4.466 (0.0022)	75.5
write	1.435 (0.0001)	3.352 (0.0007)	133.6

Table 5.3: System call latency results. All times are in microseconds. Standard deviations are listed in parentheses.

Operation	Standard (μs)	pH (μs)	% Increase
null	427.73 (00.241)	442.02 (02.919)	3.3
simple	2996.40 (09.703)	11193.32 (10.969)	273.6
/bin/sh	30158.94 (22.379)	38941.54 (15.915)	29.1

Table 5.4: Dynamic process creation latency results. *Null* refers to a fork of the current process. *Simple* is a fork of the current process plus an exec() of a hello-world program written in C. /bin/sh refers to the execution of hello-world through the libc system() interface, which uses /bin/sh to invoke hello-world. All times are in microseconds. Standard deviations are listed in parentheses.

that normally would take between 1 and 3 μs to execute. Table 5.4 shows that a simple fork requires less than 15 μs more time with pH, a 3.3% increase; a fork and an execve together, however, is almost 4 times slower. This increase comes from pH causing execve calls to take approximately 8 ms longer to run, which seems to be the time the kernel needs to load a 129K profile. Although these numbers show that pH causes a significant increase in system call latency, they are not indicative of the impact on overall system performance.

As shown by the data in the next chapter, the X-Window server is by far the heaviest user of system calls on a typical interactive workstation. To see what sort of impact pH had on system performance, I ran the *x11perf* benchmark and used it to calculate the Xmarks for lydia running the XFree86 4.0.3 Mach64 X server. The

Time Category	Standard (s)	pH (s)	% Increase
user	728.92 (0.74)	733.09 (0.17)	0.57%
system	58.19 (0.80)	80.34 (0.17)	38.06%
elapsed	798.65 (0.87)	825.18 (1.75)	3.32%

Table 5.5: Kernel build time performance. All times are in seconds. Each test was run once before beginning the measurements in order to eliminate initial I/O transients, and was followed by five trials. Standard deviations are listed in parentheses.

non-pH system got 11.4681 Xmarks, while pH-enabled system got 11.3750 Xmarks — a slowdown of only 0.81%.

Although this result is outstanding, it is somewhat misleading because the X server does not make execve calls during normal operation. Table 5.5 shows pH's impact on the compilation of the Linux kernel. As this make process invokes many different programs, it gives a better view of the impact of slow execve calls. The performance hit is especially noticeable in the system time. This category measures the time spent in the kernel, and it shows that pH requires 38% more time in the kernel. Overall this translated into a 3.32% increase in execution time — a much more acceptable value. Thus, an application that creates numerous processes and loads many different executables experiences a slowdown of less than 5%.

To put these results in perspective, it is worth comparing pH's performance with another kernel security extension that monitors system-call sequences. Ko et al. [60] implemented a generic "software wrappers" system for augmenting the behavior of system calls under FreeBSD. To show the flexibility of their framework, they implemented a module that analyzed system-call sequences. In test builds of the FreeBSD kernel, they reported a 3.47% slowdown with the Wrapper Support System (WSS) kernel module loaded, and a 6.59% overall slowdown when their `Seq_id` was loaded on top of the WSS. If we assume that the build processes of the Linux and FreeBSD kernels are comparable, the overhead of just the system-call wrappers system is ap-

proximately the same as all of pH (3.47% for WSS vs. 3.32% for pH). The addition of sequence analysis makes the software wrappers system almost twice as slow as pH on average. It seems that Ko et al. implemented a variation on full sequence analysis instead of lookahead pair analysis; therefore, it is possible that lookahead pair analysis implemented using their WSS would be more efficient than these results would suggest.

In summary, these results show that pH incurs an acceptable performance penalty of less than 5%, even when monitoring every process on a system. And, although pH's current implementation can certainly be further optimized, its efficiency is quite competitive with similar existing systems.

Chapter 6

Normal Program Behavior in Practice

Because pH detects intrusions by noticing anomalous sequences of system calls, it only detects attacks against programs for which it has a normal profile. This chapter examines some of the basic properties of these normal profiles and explore how well pH can acquire them in practice.

The first part of the chapter describes a 1-day data set in which every system call made on one computer was recorded. This data set is used to compare sequences to lookahead pairs and to evaluate different window sizes. Next a 22-day dataset from this same computer is presented, consisting of profiles and kernel log messages produced by pH. This dataset is used to explore how well pH captures normal program behavior and how often it generates false positives. pH datasets from three other hosts are then used to elaborate on this analysis. The chapter concludes with an analysis of profile diversity and a discussion of the nature of pH's false positives.

6.1 What is Normal Behavior?

When we say that a program's profile is normal, we mean that the profile represents all of the system call sequences that are likely to be observed when running the program under normal conditions. The "normal" behavior of a program or system is not a well-defined concept; indeed, our view of normal changes based on the circumstances of our observations. For example, normal behavior for a high-profile web site server might include vulnerability probes each hour, while such behavior would be extremely unusual for a web server within a protected corporate intranet.

Lacking a compelling formal definition of "normal behavior," we can instead define it operationally. From this perspective, normal behavior is behavior that is observed when we are reasonably certain no activity is occurring that requires non-routine direct human administrative observation and interaction. A system showing signs of imminent disk failure is not behaving normally. A program which has suddenly lost its configuration files is not behaving normally. A system being successfully attacked is not behaving normally. In contrast, timeouts when connecting to distant web servers and failed intrusion attempts are normal for a server on today's Internet.

In our original experiments with system-call monitoring, we explicitly exercised programs under a variety of conditions, recording the system calls produced [43]. The set of traces produced by a given program was then designated as our normal set, and a normal profile was compiled based on this data. Such "synthetic normals" are a straightforward, replicable way to study the nature of normal program behavior; however, such studies have two fundamental problems.

One problem is that it is surprisingly hard to exercise all of the normal behavior modes of any non-trivial program. Interactions with the file system, network, and other processes cause enough variations that system-call traces from two apparently identical program invocations often have significant differences. Indeed, such vari-

ations are to be expected, given the nondeterministic nature of computer networks and operating-system process scheduling. And although it may be possible to isolate a system sufficiently so that one may obtain repeatable results, it is hard to say that such a setup corresponds to the normal behavior of "real" systems.

Moreover, even if a synthetic normal captures most normal modes of behavior, it will not describe the relative frequency of these behaviors: this information depends on the program's usage environment. For example, one person may frequently use the "-l" flag to *ls* on their personal workstation to get detailed file listings, while another would never use the "-l" flag. This difference would mean that the normal profile for *ls* would differ between these two hosts. If the command *ls -l* were typed on the second machine, and if pH had a valid normal profile for *ls*, pH would signal this action as being anomalous.

Thus, with synthetic normal profiles one cannot determine whether the excluded modes of behavior are frequent or rare, and so one cannot get a true sense of false positive rate in practice. My approach here, then, is to examine the behavior of systems out "in the wild," by gathering data from production systems.

This strategy has some disadvantages. First, the experiments cannot be exactly replicated, because the conditions of the tests cannot be exactly duplicated. Data gathered from live sources may also be contaminated with actual security violations, potentially making it difficult to distinguish between true positives (genuine attacks) and false positives (other anomalous behavior). Also, such experiments are potentially dangerous and intrusive, in that pH could interfere with legitimate computer usage. These disadvantages are outweighed, however, by the prospect of discovering how well pH works in practice.

Percent	# Calls	Program	Description
31.193	73,955,823	XFree86	X-Window server
16.112	38,199,165	VMWare	virtual PC running Windows 98
13.016	30,860,620	pHmon	pH monitoring program
10.165	24,099,508	wmifs	network monitor
7.918	18,772,374	wmapm	power monitor
3.545	8,404,518	wmmixer	sound mixer
2.984	7,075,673	WindowMaker	window manager
2.900	6,875,761	asclock-gtk	clock
2.660	6,305,727	Mozilla	web browser
1.947	4,617,057	tar	tape archiver
1.918	4,547,693	wmppp	dial-up network
0.868	2,059,118	Netscape	web browser
0.852	2,019,702	sendbackup	Amanda: send backup
0.530	1,255,458	bzip2	compression program
0.401	950,471	ntpd	Network Time Daemon
0.340	805,408	rsync	synchronize directories
0.323	766,073	find	list files by properties
0.309	731,942	taper	Amanda: write tape
0.243	576,237	gnuplot	plotting program
0.182	431,669	RealPlayer	streaming media player

Table 6.1: The 20 top programs by number of system calls executed, out of 304 total, on a system monitored by pH. Note that these programs account for over 98% of the system calls executed. (Data is from the lydia 1-day training set.)

6.2 A Day in Detail

A normally functioning system makes a huge number of system calls. To better understand the nature of these calls, I recorded all of the system calls made by my home computer (lydia) for the period of one day, starting on August 17th, 2001 at 9 PM. During this day I performed typical tasks such as reading email, surfing the web, editing text files, and listening to streaming audio. In addition, background daemons and cron jobs were running, including a daily backup performed every weeknight at 2:15 AM. Over the course of these 24 hours, lydia made 237,224,596 system calls

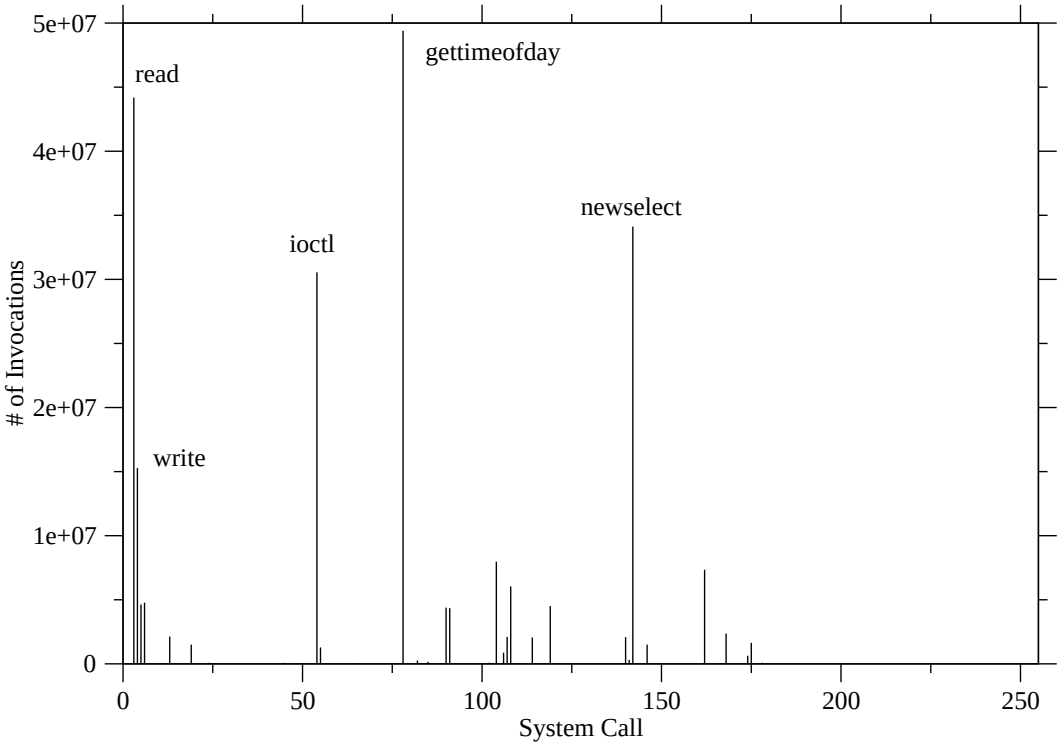

Figure 6.1: Frequency of different system calls. (Data is from the lydia 1-day data set.)

while running 304 different programs.

Looking at Table 6.1, it is remarkable to see how a few programs make almost all of the system calls. Similarly, Table 6.2 and Figure 6.1 show how just a few system calls also dominate. The dominating programs clearly influence which system calls are most frequent: for example, the most frequent system call, gettimeofday, was called over 17 million times by XFree86, the X server.

A simple measure of frequency doesn't capture the differing complexity of these programs, though. One way to get a handle on this is to look at the sequence and lookahead entropy, as explained in Chapter 4. Figure 6.2 shows how entropy increases exponentially as we consider the joint distribution of more adjacent system

Percent	# Invocations	Call #	Call Name
20.83	49,403,645	78	gettimeofday
18.62	44,182,442	3	read
14.38	34,119,064	142	newselect
12.88	30,552,704	54	ioctl
6.44	15,277,874	4	write
3.36	7,968,458	104	setitimer
3.09	7,341,818	162	nanosleep
2.55	6,046,034	108	fstat
2.01	4,771,191	6	close
1.95	4,625,396	5	open
1.90	4,505,801	119	sigreturn
1.85	4,386,163	90	mmap
1.83	4,342,443	91	munmap
0.99	2,358,735	168	poll
0.90	2,129,580	13	time
0.88	2,096,626	107	lstat
0.87	2,075,123	140	llseek
0.87	2,059,081	114	wait4
0.69	1,636,532	175	rt_sigprocmask
0.63	1,503,479	19	lseek

Table 6.2: The top 20 most frequent system calls. There were 139 different system calls that were called at least once. Note that these 20 calls make up over 97% of all system calls. (Data is from the lydia 1-day training set.)

calls; it also shows a significant difference in entropy for programs making a similar numbers of calls. Simple monitoring programs such as *wmapm*, *wmifs*, and *pHmon* repeatedly make the same basic set of system calls as they periodically update their view of the system's state. Since these patterns are repetitive, they can be captured on average by short descriptions; thus, the sequence entropy for these programs is relatively low. In contrast, we have VMWare, a program that emulates an entire computer system, allowing "guest" operating systems to run as a Linux process. On this machine, VMWare was used to host a guest system running Microsoft Windows 98. Since VMWare is using all of the resources of a complete operating system, it is

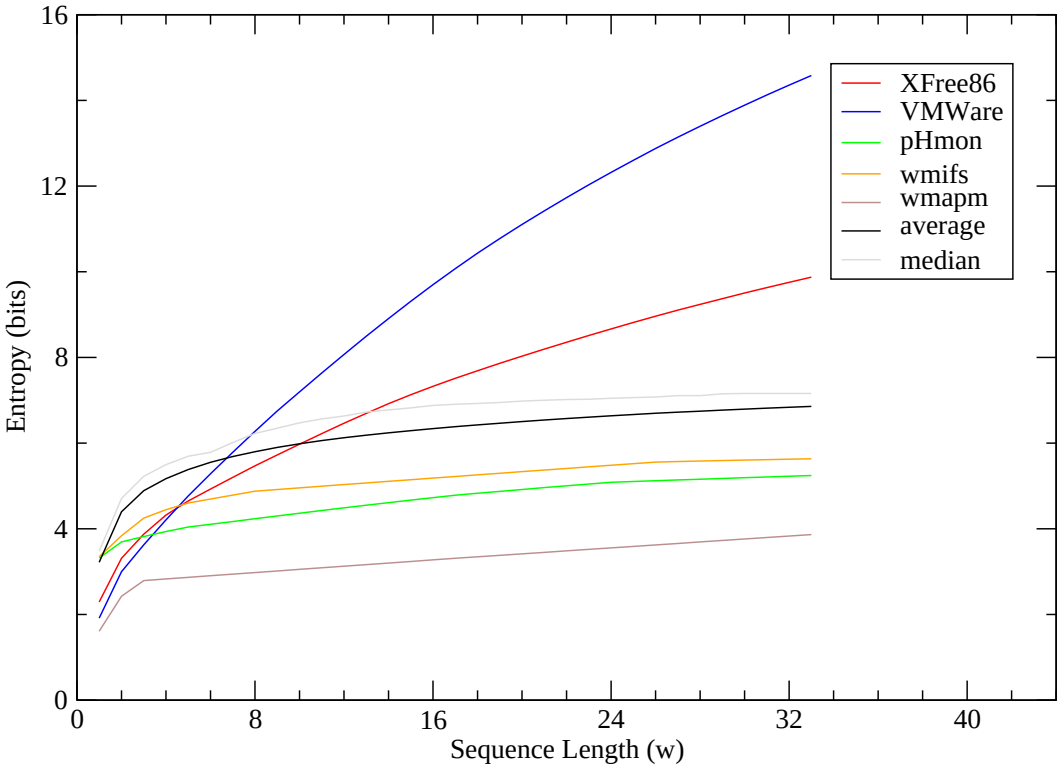

Figure 6.2: A graph of $H(\mathcal{S}_w)$ (sequence entropy), for the top 5 programs by number of system calls, along with the average and median for all 304, vs. different sequence length (w). (Data is from the lydia 1-day training set.)

no surprise that it makes many system calls, and that the pattern of these system calls varies significantly. This complexity is the source of VMWare's large sequence entropy.

Oddly enough, however, the flat curves in Figure 6.3 show that the lookahead entropy is rather consistent. This pattern seems to imply that there isn't a natural distance at which to look for correlations between system calls, at least for windows of size 33 or less. This result corroborates earlier past work by Kosoresow and Hofmeyr on sendmail behavior [63].

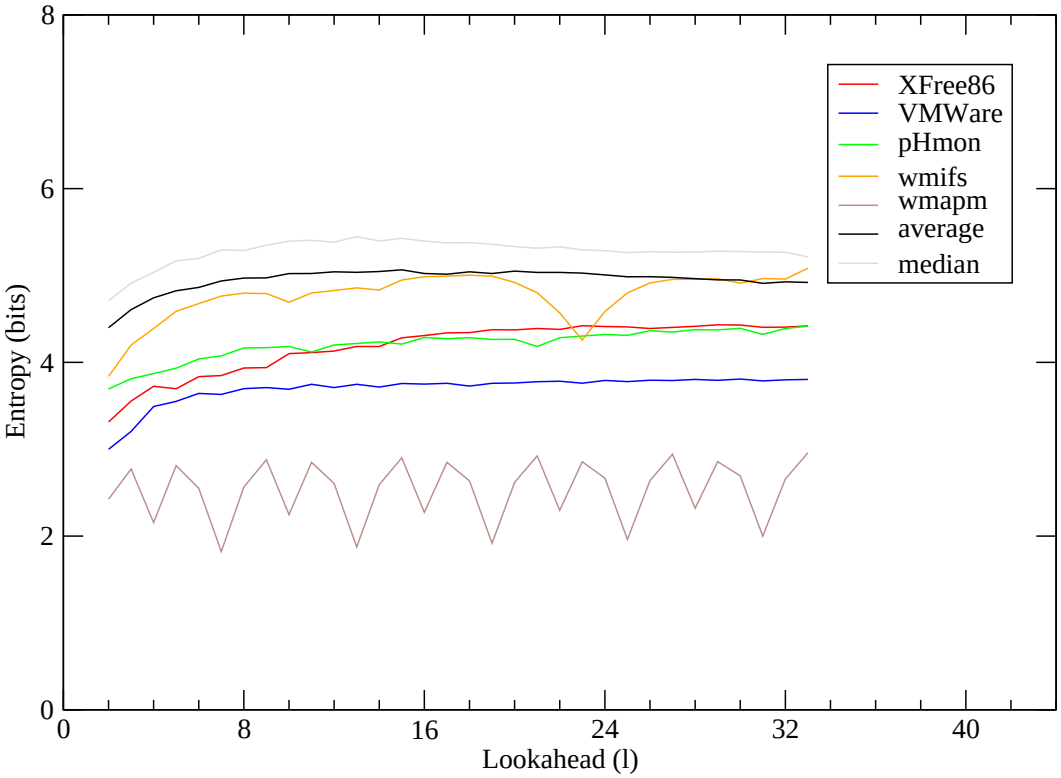

Figure 6.3: A graph of $H(\mathcal{L}_l)$ (lookahead pair entropy), for the top 5 programs by number of system calls, along with the average and median for all 304, vs. different sequence length (w). These entropy values for the pairwise distribution show how lookahead pairs of different separations roughly give the same amount of information about a program's behavior. (Data is from the lydia 1-day training set.)

6.3 Choosing a Method & Window Size

The smoothness of the average and median curves in Figures 6.2 and 6.3 imply that there is no specific window size beyond which there is no correlation. This conclusion is reinforced by Figure 6.4. In this graph, we see how much normal behavior was seen on average before we saw 50%, 90%, or 95% of the lookahead pairs or sequences[1]. The smoothness of these curves also suggest that there isn't a natural window size.

[1]Note that for the lookahead values, this fraction includes all smaller lookahead values: at $w = 4$, the lookahead curves include the fraction of lookaheads for l=2, 3, and 4.

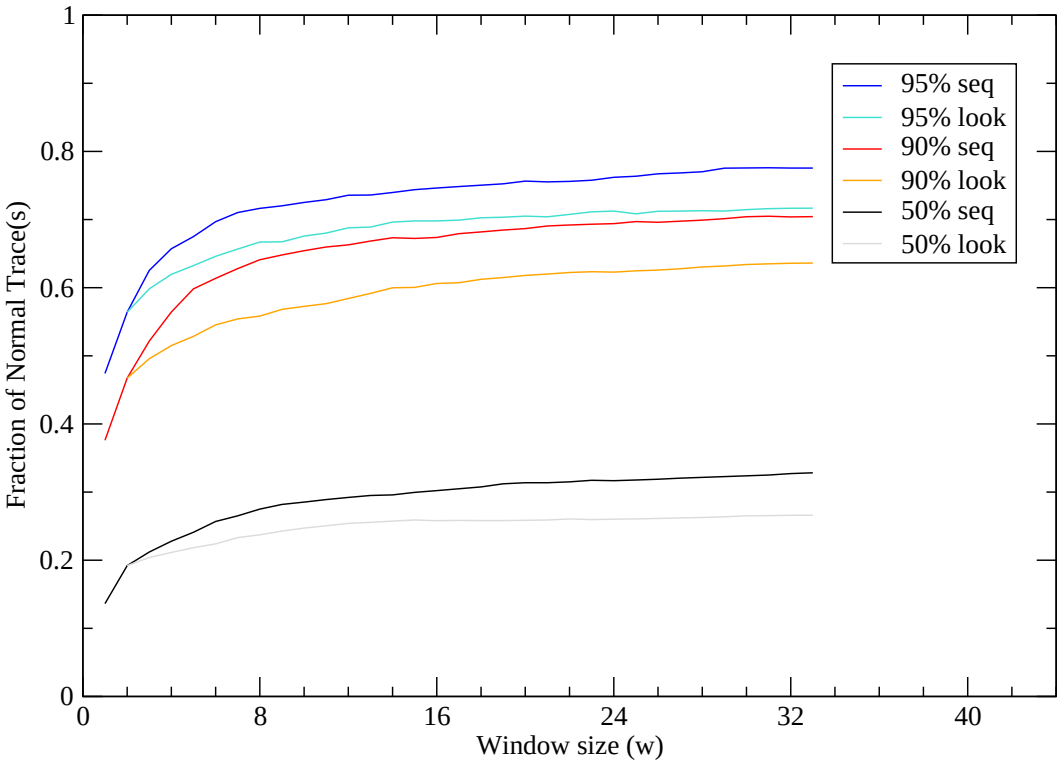

Figure 6.4: This graph shows the fraction of normal behavior that was seen, on average, before a percentage of patterns in a given profile were seen. For example, the 50% look curve at $w = 9$ shows that on average we had to observe .243 (24.3%) of the normal trace to get 50% of a profile's lookahead pairs for a window size of 9. Data comes from the lydia 1-day data set.

Even if there isn't a natural window size, however, there is one simple trade-off that should be considered: the larger the window, the more potential anomalies, but also the more storage and training required. A larger window produces more anomalies because any deviation from previously-seen patterns will create new sequences or lookaheads in proportion to the length of the window. Some of these patterns may already be in the profile, but with the greater set of possibilities in a larger window, it is more likely that some of these patterns will not be present in the normal profile. At the same time, storage requirements grow linearly for lookahead pairs and (potentially) exponentially for sequences as one increases the size of the

Parameter	Value
loglevel	3
log_syscalls	0 (1)
log_sequences	0
delay_factor	1 (0)
normal_factor	48 (128, 64)
normal_wait	172800
anomaly_limit	30
tolerize_limit	12
abort_execve	10 (0)

Table 6.3: The parameter settings for pH during the lydia 22-day experiment. Note that there were a few changes during the run. *log_syscalls* was 0 except for August 17-18th, when it was 1, causing every system call to be logged to disk. *delay_factor* and *abort_execve* were temporarily set to 0 on August 13th, from 8:30-10 PM. Also, *normal_factor* was 128 from August 10-15th, 64 from the 15-17th, and 48 from August 17th to September 1st.

window. (See Section 4.3 for a more detailed explanation.)

To maximize speed and efficiency and to minimize storage overhead and code complexity, pH uses lookahead pairs to observe program behavior. Because we can fit eight bit arrays in a 256 by 256 byte array, pH uses a window size of 9. Anything larger than this size would require a doubling of storage requirements; anything smaller means wasted space and a lower chance of detecting anomalies. In the past we have used smaller window sizes, particularly size six; the curves of Figure 6.4 show that on average, we will pay a small penalty in training time for this increase in window size. In return, the larger window allows pH to be more sensitive to anomalies.

6.4 A Few Weeks on a Personal Workstation

One motivation for implementing pH was to allow large amounts of system behavior to be observed over an extended period of time. For example, the single day of system calls used in the previous section takes up over 803M of highly compressed (bzip2) storage. In contrast, profiles and logs for three weeks of running pH take up 538K similarly compressed (70M uncompressed). Clearly much information has been lost; however, what remains provides insight into pH's perception of normal system behavior.

The data set used in the following sections was gathered from August 10th at 3:05 PM to 2:20 PM on September 1st, 2001. The test system again was my home computer, lydia, which was used on a daily basis during this time. This computer is continuously connected to the Internet through a shared T-1 line with a static IP address, and sits behind a simple NAT firewall which routes outside SSH connections to it. Thus, although it is a home machine, it is also a full-fledged Internet workstation. There is a risk in using myself as a test subject, in that my knowledge of pH influences how I use the computer; the advantage, though, is that I can correlate my actions with the actions of pH.

Table 6.3 shows the parameter settings that were used for pH during this time. These settings were kept consistent with a few exceptions. On August 13th, between 8:30 and 10 PM, both $delay\_factor$ and $abort\_execve$ were set to 0 to deal with a rash of false positives; this incident is detailed in Section 6.6. In addition, $normal\_factor$ was changed from its initial value of 128 to 64 on August 15th (11:25 AM), and then to 48 on August 17th (9:53 PM). These settings were changed to make pH freeze profiles with less data, and as explained below, these changes did accelerate the normal classification of several programs — at the cost of additional false positives.

This 22-day data set contains profiles for 528 programs which have in total made

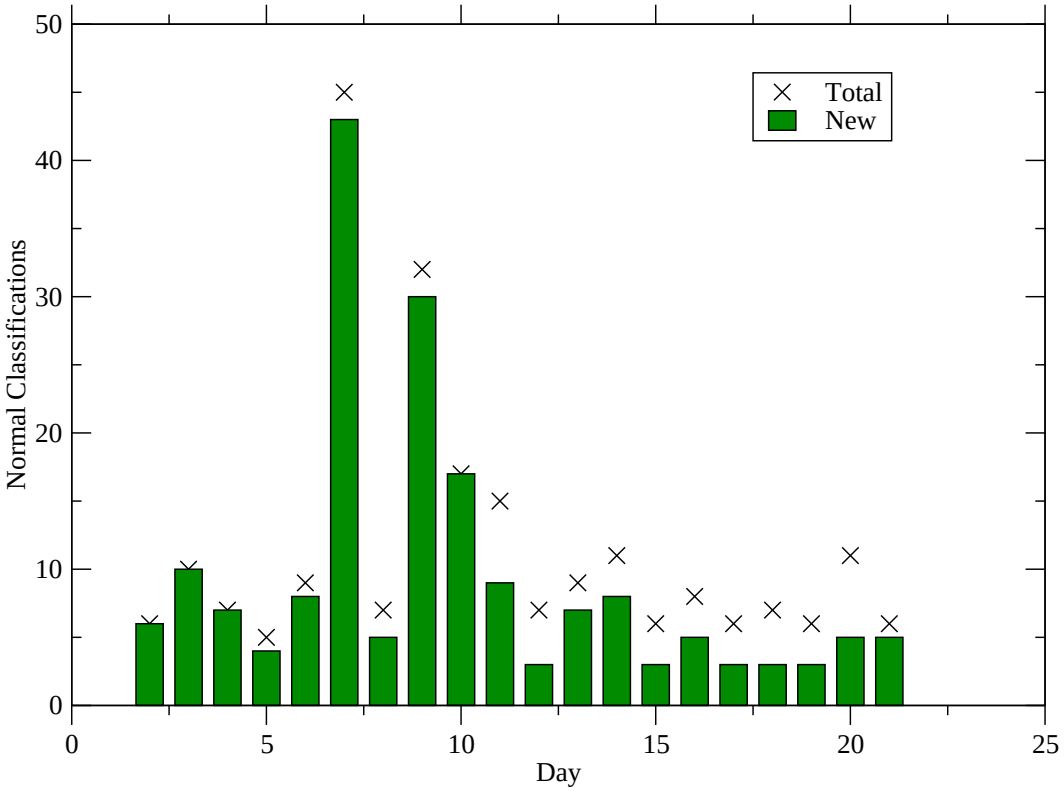

Figure 6.5: Normal classifications per day for the 22-day lydia data set. The crosses mark the total number of normal classifications, while the bars show the number of new normal classifications.

5,772,818,962 system calls. Table 6.4 shows the top 20 programs by system call for this data set. Although the numbers are significantly larger, the list is otherwise similar to that of the 1-day data set.

6.5 Normal Monitoring

As described in Section 5.3, pH uses a two-part algorithm to decide when it may classify the training data for a program as representing normal program behavior. As the first part of this process, pH "freezes" a profile when a sufficient number of

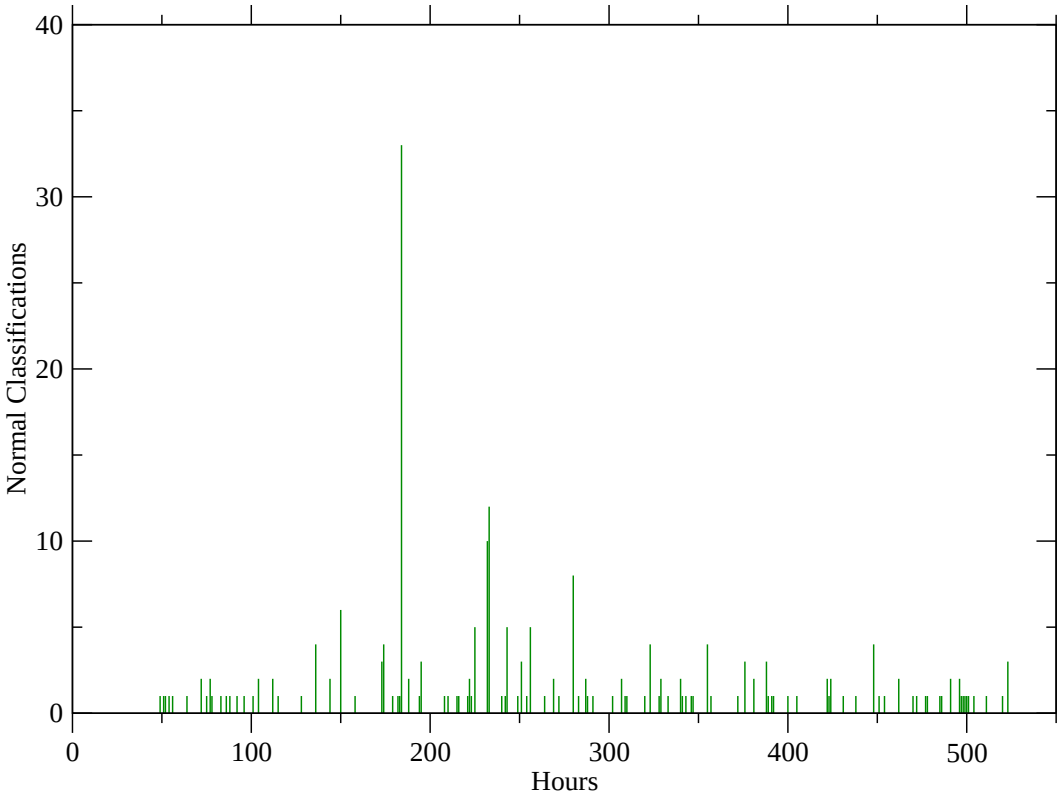

Figure 6.6: Normal classifications per hour for the 22-day lydia data set.

calls have been made without generating any new lookahead pairs. (This amount is controlled by the $normal\_factor$ ratio.) If a profile remains frozen for two days ($normal\_wait$ seconds), then the training data is copied to the testing array, and the profile is marked as normal.

During the 22 days of monitoring, pH classified 184 programs as normal in 230 distinct events (some programs were classified as normal multiple times). Figure 6.5 shows how many programs were classified as normal each day. Figure 6.6 shows the same but on an hourly basis. These events are relatively evenly distributed except for a few large peaks. The largest peak, at day 7 and hour 184, corresponds to 7-8 AM on August 18th. Daily, weekly, and monthly cron jobs get run at 7:30 AM, and this peak corresponds to the first time that they were run after $normal\_wait$ was

reduced to 48 on August 17th. There are smaller, but higher-than-average peaks at 24 hour intervals after this.

This pattern of peaks shows that pH is good at profiling the behavior of small, simple programs that are run frequently. In Table 6.5, we can see that the two directories with the highest percentage of normal profiles correspond to programs that are run at the same time every day: The programs in /etc/cron.daily are run each day at 7:30 AM to perform tasks such as log rotation and updates to the *locate* database; the programs in /usr/lib/amanda are part of the Amanda client/server backup system, which runs each weekday night at 2:15 AM on my home network.

Looking at the broader system, this pattern continues to hold. Complicated, frequently used programs such as Emacs and Mozilla are never classified as normal. Small, infrequently-called programs such as those in /etc/init.d (which are called on system startup/shutdown) also rarely become normal. In contrast, simple programs that are primarily called by scripts, such as *cut*, *mesg*, and *file* tend to settle down. Large programs can sometimes be classified as normal: VMWare was marked as normal near the end of the testing period on August 31st, and subsequently it only generates a few false positives per run (which are barely noticeable). It is surprising that pH could do this. It is true that I primarily use VMWare to run Quicken under Windows 98 and to make sure there are no glaring bugs in new versions of pH; however, VMWare is still a rich, complicated program.

System daemons are also often classified as normal. During this run, programs such as *named* (day 16), *inetd* (day 13 and 17), and *dhcpd* (day 21) were classified as normal. Unfortunately, other less-used programs such as *sshd* were not classified as normal, and so pH did not provide any protection directly for these programs. This limitation does not mean that pH would provide no protection, since novel uses of other protected programs would likely cause an attacker difficulty. Since pH has not yet responded to an attack "in the wild," however, this claim is still unproven.

Percent	# Calls	N/F	Program
30.29	1,748,920,800		XFree86
14.02	809,861,982	F	wmifs
12.77	736,981,914		pHmon (inst)
8.75	504,996,876	NF	wmapm
6.71	387,217,845	F	Mozilla
6.17	355,977,607	F	pH-print-syscalls (inst)
2.95	170,085,293	N	wmmixer
2.49	144,009,812		WindowMaker
2.41	139,150,260	F	asclock-gtk
2.12	122,290,709	N	wmppp
2.02	116,662,846	N	VMWare
1.58	91,336,269		pH-print-syscalls (dev)
1.20	69,159,315		Acrobat Reader
1.15	66,587,812		tar
0.61	35,419,072		sendbackup
0.54	31,321,283		pHmon (dev)
0.52	30,023,696		Netscape
0.36	20,579,896	F	/usr/bin/top
0.31	18,023,251		/usr/sbin/ntpd
0.28	16,129,866		/usr/bin/find

Table 6.4: The 20 top programs by number of system calls executed, out of 528 total. The N/F column indicates whether the corresponding profile was normal and/or frozen at the end of the experiment. The dev/inst tags refer to the program being either installed on the system in /usr/local/bin, or residing in my development home directory. Note that these programs account for over 97% of the system calls executed. (Data is from the lydia 22-day training set.)

Directory	# Profiles	# Normal	% Normal
/usr/bin	209	56	26.8
/bin	44	21	47.7
/usr/sbin	42	18	42.9
/etc/init.d	38	2	5.3
/sbin	24	5	20.8
/usr/X11R6/bin	23	10	43.5
/etc/cron.daily	19	19	100.0
/usr/lib/amanda	12	9	75.0
/etc/menu-methods	9	0	0.0
/usr/local/bin	8	2	25.0
/etc/cron.weekly	8	0	0.0
/usr/lib/netscape/base-4	3	1	33.3
Total	439	143	32.6
System Total	528	161	30.5

Table 6.5: The number of normal profiles in 12 system directories. (Data is from the lydia 22-day data set.)

	Per Hour	Per Day	Total
Anomalies	2.01	48.3	1061
Unique Anom. Programs/hour	0.247	5.92	130
Unique Anom. Programs/day	0.195	4.69	103
Unique Anom. Programs	0.125	3.00	66
Tolerizations	0.0929	2.23	49

Table 6.6: False positives statistics, in terms of anomalies, unique anomalous programs, and tolerization events. Note how unique anomalous programs are listed by hour, day, and for the whole data set, since the number of unique programs varies depending on the granularity of the time bins. There were 21.968 days, or 527.24 hours in the data set. (Data is from the lydia 22-day data set.)

6.6 False Positives

In past work, we have estimated the number of false-positives that a system-call monitoring system might generate [49, 116]. With pH, we now have our first chance to measure a false-positive rate in practice.

Table 6.6 shows that the false-positive rate of pH varies greatly depending upon how you measure it. If we look at the raw number of anomalies generated by pH, we get a bit over two per hour. If we consider the number of unique anomalous programs per day, there are fewer than five false positives per day. But probably the most important statistic is the frequency of user tolerization events, which come out to two per day. As explained in Section 5.5, user tolerization events occur when the user decides that a given program is being wrongly delayed and tells pH to stop monitoring that program for anomalies. The normal flag for the corresponding profile is set back to 0, and training resumes from where it left off. Because such events are initiated by a person, they represent the false-positive rate as perceived by the user or administrator.

Figure 6.7 shows that low numbers of anomalies often do not lead to intervention,

Cause	MaxLFC	Program
anacron	13	/etc/cron.daily/0anacron
(weekly cron)	18	/etc/init.d/ntp
	18	/usr/sbin/anacron
	19	/usr/sbin/logrotate
Amanda	20	/usr/lib/amanda/driver
(backup)	21	/usr/sbin/inetd
User actions	10	/bin/chmod
	10	/bin/cp
	10	/bin/egrep
	14	/bin/netstat
	10	/bin/run-parts
	10	/bin/zcat
	10	/sbin/route
	15	/usr/lib/netscape/base-4/netscape-remote
	10	/usr/bin/AbiWord
	16	/usr/bin/basename
	12	/usr/bin/glib-config
	10	/usr/bin/top
	13	/usr/local/bin/pH-admin
	10	/usr/local/bin/pH-ps
	13	/usr/sbin/nmbd

Table 6.7: False positives: The 21 programs in the lydia 21-day data set that had a maxLFC value equal or greater to the *abort_execve* threshold of 10. Note that most of these false positives, the "user actions," were caused by my using these programs in new ways. Thus, pH was acting properly in these situations.

while a large clumping of anomalies is almost always followed by one or more user tolerization events. One disturbing trend is that there appears to be more frequent clusters of low-level anomalies as time goes on. This increase is due to the greater number of program profiles that pH has classified as normal. Clearly, pH makes some mistakes.

These mistakes are more evident if we look at which programs are generating anomalies. Table 6.7 shows the 21 programs that had a maxLFC score of 10 or

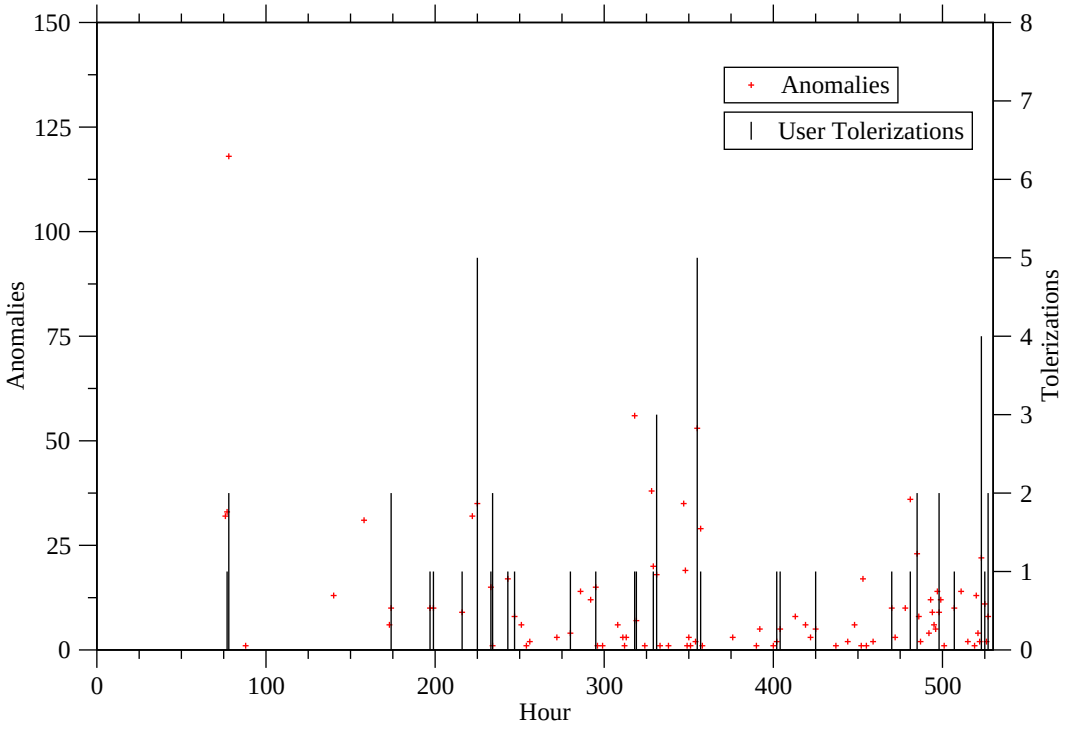

Figure 6.7: The number of anomalies and user tolerization events per hour for the 22-day lydia data set.

higher. Because *abort_execve* was set to 10, these programs, in addition to being (potentially) delayed for over twenty minutes, would also have been forbidden from

No Anom. for	Tol. After	MaxLFC	Program
85.7h	8s	6	/bin/ln
85.0h	2s	3	/usr/bin/tail
302.7h	1s	5	/usr/sbin/oidentd
6.5h	43.3h	1	/usr/X11R6/bin/XFree86

Table 6.8: The four programs which exceeded the *anomaly_limit* threshold during the 22-day lydia experiment. The first column denotes the time between the program's profile being classified as normal and the program's first anomaly. The second column contains the time from the first anomaly to when the *anomaly_limit* threshold of 30 was exceeded.

making execve calls. This chart shows that most of these false positives were the direct result of user actions, while the rest came from programs that run periodically, either on a nightly or weekly basis.

Whether it is caused directly by a user or is run automatically, pH has problems with programs that are run frequently but which occasionally (or periodically) change their behavior. pH can be taught about periodic events by increasing the time of *normal_wait* to include the events in question, or through the manual triggering of these events. User-caused false positives are harder to address and probably require more sophisticated heuristics, possibly in a userspace daemon. Such possibilities are examined in the discussion.

pH can sometimes recover from its mistakes through the *anomaly_limit* mechanism, which allows pH automatically to tolerize programs that are repeatedly delayed for short periods of time. Table 6.8 shows how pH automatically tolerized four programs. The first three went significant periods of time without generating any anomalies; then, when several instances of the same programs were invoked with each incurring only a few anomalies, pH tolerized them within a matter of seconds. The last program, the XFree86 X-server, incurred anomalies 6.5 hours after starting normal monitoring, and had repeated, but short, delays for almost two days. Eventually the anomaly count for its profiles exceeded *anomaly_limit* and it was tolerized.

Host	lydia	badshot	jah	USN
Location	apartment	UNM CS	UNM CS	K-12 school
OS	Debian 3.0	Debian 2.2	Debian 2.2	Debian 2.2
Workload	devel./prod.	productivity	mail/web server	web server
Start Time	10 Aug 2001	12 Sep 2001	09 Sep 2001	19 Oct 2001
End Time	01 Sep 2001	07 Oct 2001	21 Nov 2001	08 Jan 2002
Total Days	21.97	24.82	72.74	81.65
Reboots	4	2	3	0

Table 6.9: Host and Data details on the 4 profile data sets.

Parameter	Value
loglevel	3
log_syscalls	0
log_sequences	0
delay_factor	1
normal_factor	128
normal_wait	604800
anomaly_limit	30
tolerize_limit	12
suspend_execve	10
susp_exec_time	172800

Table 6.10: The parameter settings for pH on jah, badshot, and USN for the experiments listed in Table 6.9.

6.7 Normal Behavior on Other Hosts

To see whether the results from my home computer were typical, I monitored pH's on three other hosts. The datasets collected from these machines (along with the lydia 22-day dataset) are listed in Table 6.9. Badshot is used by one person primarily for personal productivity tasks such as email, web access, and document creation. Jah is the UNM Computer Science Department's mail and web server; in addition, users can use jah for remote interactive sessions. USN is the webserver www.usn.org,

	lydia	badshot	jah	USN
Total Days	21.97	24.82	72.74	81.65
Normal-min Days	2	7	7	7
Total Profiles	528	279	823	213
Normal Profiles	161	72	105	62
% Normal Profiles	28.6%	25.8%	12.8%	29.1%
New Normal Events	184	84	133	71
Total Normal Events	230	85	161	88

Table 6.11: Normal profile summary for the lydia, badshot, jah, and USN datasets listed in Table 6.9.

a site belonging to the University School of Nashville, a kindergarten through 12th grade school in Nashville, TN. Note that two of the machines, USN and jah, are web servers, and two of the machines, badshot and jah, are located in the UNM Computer Science department. Also, all three machines are running the same Linux distribution, Debian GNU/Linux 2.2. The same 2.2.19 Linux kernel patched with pH was installed on all three machines. pH's parameter settings for these three experiments are listed in Table 6.10.

Table 6.11 compares the ability of pH to capture normal profiles on the four tested hosts. Note that at the end of the test run, pH had only classified 12.8% of jah's profiles as normal, in contrast to the 25% or more fraction on the other three hosts. This discrepancy was probably due to multiple users each using different programs on jah for brief periods of time. In contrast, USN saw relatively little interactive usage, while badshot and lydia were each primarily used by one person.

The graph in Figure 6.8 shows the pattern of new and total normal classifications on USN during the test period. (Compare this graph with Figure 6.5.) The pattern on the other two hosts is similar. Note how the number of new normal profile classifications tapers off over time. The less consistent pattern of total classifications shows that pH periodically re-classifies some profiles as normal.

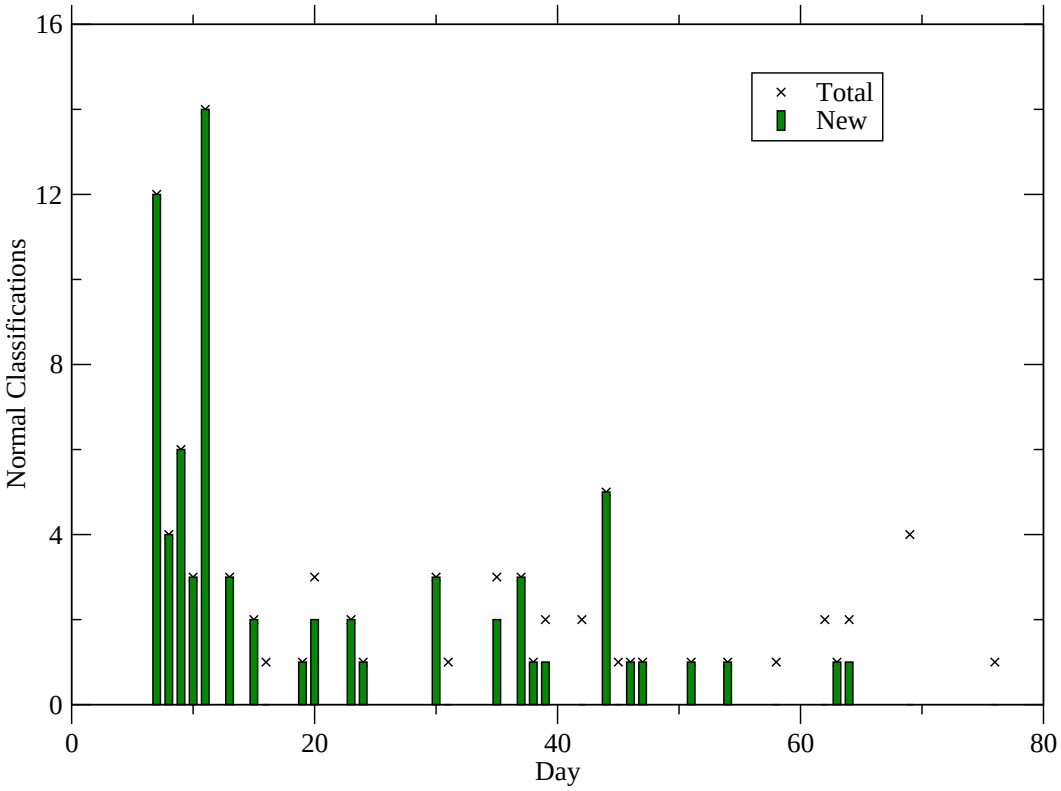

Figure 6.8: Normal classifications per day for the USN data set. The crosses mark the total number of normal classifications, while the bars show the number of new normal classifications.

Table 6.12 compares the false-positive rates for these four datasets. To account for differences in training time, each of the rates are reported in terms of "response days," or the number of days during which pH could have mounted a response. This number is the total number of days minus *normal_wait* days (2 for lydia, 7 for the other three hosts). Note that USN only required one user tolerization every five days on average. These results imply that program behavior on USN was particularly regular; this observation is reasonable since USN was not generally used for email, web surfing, or other interactive activities. In contrast, jah was used both as a server and for interactive use; thus, its behavior was more variable.

	lydia	badshot	jah	USN
Response Days (RD)	19.97	17.82	65.74	74.65
Anomalies	1061	557	1435	469
Auto Tolerizations	4	5	22	5
User Tolerizations	49	8	40	15
			(24)	
Anomalies/RD	53.13	31.26	21.83	6.28
Auto Tolerizations/RD	0.20	0.28	0.33	0.07
User Tolerizations/RD	2.45	0.45	0.61	0.20
			(0.37)	

Table 6.12: False positives for the lydia, badshot, jah, and USN datasets listed in Table 6.9. "Response Days" is the total days minus *normal_min* days, and so is the number of days during which pH could have responded to anomalies. "Auto Tolerizations" are tolerizations caused by profiles having more than *anomaly_limit* (30) anomalies, and "User Tolerizations" are tolerizations caused by direct user intervention. The numbers in parentheses exclude erroneous manual tolerizations performed on jah by a CS systems administrator.

One surprising detail in this table is the number of automatic tolerizations, especially on jah. Apparently variations in jah's behavior caused pH to delay programs that were behaving normally; the *anomaly_limit* mechanism detected these problems and prevented pH from permanently degrading the behavior of the these programs without requiring user intervention.

6.8 Profile Diversity and Complexity

One somewhat surprising observation is that program profiles are quite diverse. Consider the graph in Figure 6.9. Here we see that relatively few lookahead pairs are shared by most profiles, and that most lookahead pairs belong to only a few programs. More specifically, there are only 2314 (6.76%) lookahead pairs that are in 10% or more of the profiles (53 out of 528). Further, 12,908 of the lookahead pairs (out

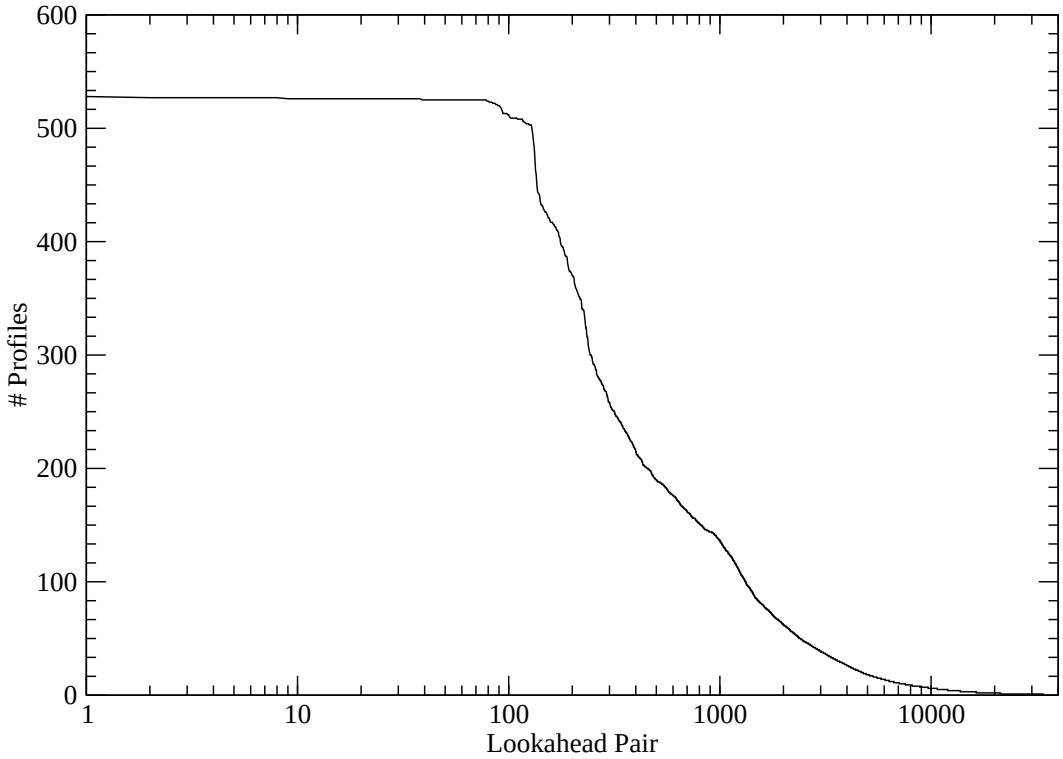

Figure 6.9: This graph shows the number of profiles containing each lookahead pair sorted by frequency. Note that the X axis scale is logarithmic. There are 528 profiles which in total contain 34227 distinct lookahead pairs. The data is from the 22-day lydia data set.

of 34,227 pairs total, 37.7%) belong to only one profile. This diversity is remarkable, considering that most programs at a bare minimum rely on the standard C library, either directly or through an interpreter.

Another way to see profile diversity is to look at the number of lookahead pairs per profile. Figure 6.10 shows the number of lookahead pairs and the corresponding number of system calls per profile. Clearly the raw number of system calls seen does not directly correspond to the complexity of the resulting profile. Table 6.13 shows this even more starkly. Programs such as Emacs and StarOffice generate a remarkable number of unique lookahead pairs while making a relatively modest

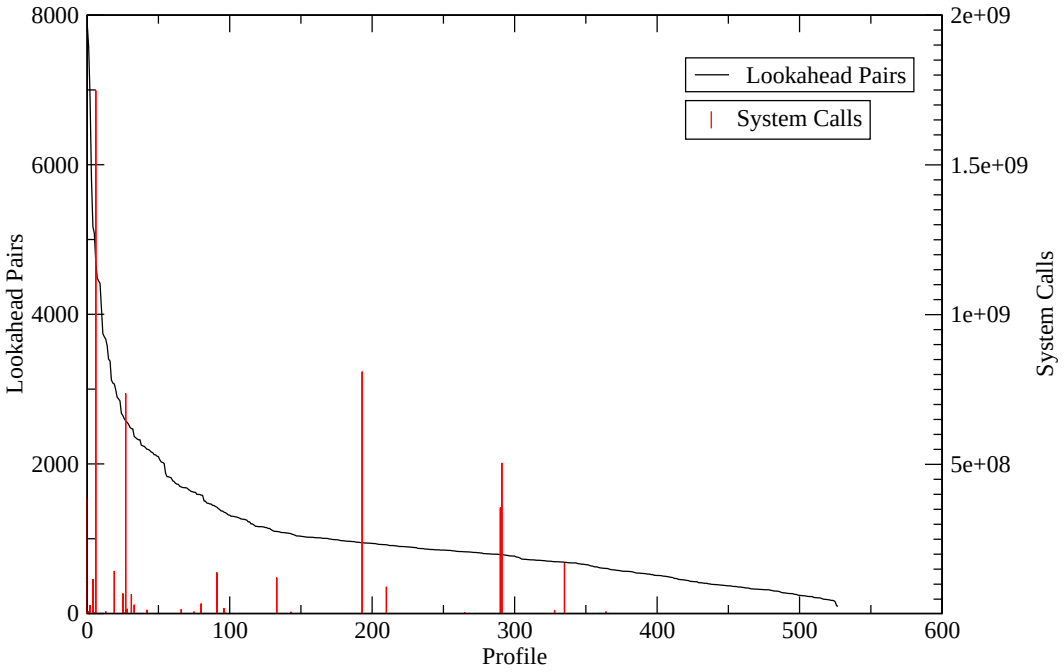

Figure 6.10: This graph shows the number lookahead pairs and system calls observed per profile, sorted by the number of lookahead pairs. This graph shows that programs with the largest lookahead pair profiles do not necessarily make the most system calls. There are 528 profiles which in total contain 34227 distinct lookahead pairs. The system call peaks at profile 194, 291, and 292 correspond to *wmifs*, *pH-print-syscalls*, and *wmapm*, respectively. The data is from the 22-day lydia data set.

number of system calls. The complexity of StarOffice is particularly notable since I use it infrequently.

Although these results are suggestive, they do not give a clear indication of how variable profiles are on a given host. One way we can quantify the diversity of profiles on a host is to define a measure of profile similarity. I define profile similarity as being the following ratio:

$$\text{profile similarity} = \frac{\#\text{ lookahead pairs in intersection}}{\#\text{ lookahead pairs in union}}$$

Although profile similarity can be computed for several profiles at once, pairwise profile similarity is the easiest to understand. For example, consider two profiles, A

System Calls	N/F	Pairs	Program
387217845	F	7883	Mozilla
8933028		7577	Emacs
30023696		6913	Netscape
73341		5797	StarOffice
116662846	N	5169	VMWare
6194085		5086	kdeinit
1748920800		4707	XFree86
2616915		4487	bash
93659		4447	smbd
2455989		4412	RealPlayer
3820209		4071	exim
71723		3741	konqueror
28731		3701	lpd
8770368		3666	xmms
1315093		3584	sshd
455286		3400	kspread
3394680		3376	aviplay
40310		3122	cvs
31865		3081	gdm
144009812		3069	WindowMaker

Table 6.13: The twenty top programs by number of lookahead pairs. The second column indicates whether the profile was frozen or classified as normal. Note how the top five programs correspond to packages that are generally considered to be complex.

and B, where A has 500 lookahead pairs, and B has 1000 lookahead pairs, and A is a strict subset of B. The similarity of A and B would then be 500/1000, or 0.5.

To measure the diversity of profiles on a given host, we can compute the similarity of every pair of profiles. The average of these similarities gives us the expectation of how similar any two profiles will be on a host. As Table 6.14 shows, any two profiles are on average 1/5 to 1/4 similar. As the large standard deviations indicate, though, there is much variation in this measure.

A more interesting measure is the similarity of programs between hosts. Table

Host	# Profiles	Comparisons	Similarity
lydia	528	139,128	0.228 (0.148)
badshot	279	38,781	0.213 (0.144)
jah	823	338,253	0.232 (0.159)
USN	213	22,578	0.248 (0.182)

Table 6.14: The average profile similarity for the four tested hosts. The pairwise profile similarity was computed for each pair of profiles on a given host. Standard deviation is given in parentheses.

6.15 shows the result of computing the similarity of profiles for the same binaries. Thus, the comparison between jah and USN compared jah's *ls* profile to USN's *ls* profile, jah's *emacs* to USN's *emacs*, and so on. The first number in each box is the average similarity of profiles for the same program on different hosts. The second number is the standard deviation of this average. The third number in each box lists

	lydia	badshot	jah	USN
lydia	1.000 (0.000) 528	**0.461** (0.124) 197	**0.451** (0.116) 246	**0.446** (0.124) 144
badshot	**0.461** (0.124) 197	1.000 (0.000) 279	**0.754** (0.201) 226	**0.785** (0.211) 142
jah	**0.451** (0.116) 246	**0.754** (0.201) 226	1.000 (0.000) 823	**0.730** (0.209) 173
USN	**0.446** (0.124) 144	**0.785** (0.211) 142	**0.730** (0.209) 173	1.000 (0.000) 213

Table 6.15: The average host similarity for the four tested hosts. The pairwise profile similarity was computed for each profile that two hosts have in common. The first number (in bold) is the average profile similarity. The second number (in parentheses) is the standard deviation for this average. The third is the number of profiles that the two hosts have in common.

the number of profiles that the two hosts have in common.

One thing that is apparent from this table is that lydia's profiles appear to be significantly different from the other three hosts. This difference is probably due to the fact that lydia was running a pre-release of Debian 3.0, while the other three hosts were running Debian 2.2. Another observation is that the profiles from two hosts are at most 0.785 similar (for USN and badshot). The standard deviations are as high as 30% of the similarity value, so some programs are very similar between two hosts; nevertheless, this table shows that pH's profiles do differ from machine to machine based on their configuration and usage, and not just based on differences in program versions.

These results show that pH's definition of normal behavior varies from program to program and from machine to machine. This diversity offers the possibility that a successful attack against one machine may not work on another, even if both are running the same program binaries. More work needs to be done, however, to determine whether this diversity adds much protection in practice.

6.9 *suspend_execve* Issues & Longer-Term Data

Up to this point, all of the results in this chapter have come from pH running with the *suspend_execve* parameter set to 10. This value is actually rather high, and means that each system call is being delayed for 10 seconds before attempted execve calls trigger a two-day delay. In practice, such a value means that the *suspend_execve* response is almost never invoked.

This setting was chosen to minimize the likelihood of pH causing significant problems for users. Every time the *suspend_execve* response is used, a person needs to either kill or tolerize the affected process; since I couldn't guarantee that someone

Total Profiles	1467
Normal Profiles	80
% Normal Profiles	5.4%
Response Days	14.77
Anomalies	172
Tolerizations	23
Anomalies/RD	11.65
Tolerizations/RD	1.56

Table 6.16: Data from lydia with *suspend_execve* set to 1. pH's other settings were the same as those listed in Table 6.10.

would be able to interact with pH on a timely basis, it seemed wise to keep the *suspend_execve* threshold high. The results of Chapter 7, however, suggest that pH needs to have *suspend_execve* set to 1 in order to catch buffer overflows and backdoors.

To see how practical it is to run pH with lower *suspend_execve* values, I have run pH on lydia with *suspend_execve* set to 1 from September, 2001 through March 24, 2002. pH has not run without interference during this entire period, though. A C library upgrade in late December required that all normal profiles be tolerized. lydia was not in use for three weeks in February. Other kernels were run to diagnose hardware problems. Altogether, though, pH was used for approximately two months after the December tolerizations.

Table 6.16 presents data from the last two weeks of this extended pH run. This data set had only half of the normal profiles of the earlier lydia dataset (80 vs. 161); this number, however, is comparable to the number of normal profiles on the other three hosts, even though the total number of profiles is much larger (1467 vs. 528). Although relatively few profiles are normal, this set has one particularly notable member: the XFree86 X-Server, version 4.1.0. It was classified as normal ten days before the end of the run with 5551 lookahead pairs (up from 4707), and in those ten

days it made over one billion system calls. Only one of these billion system calls was anomalous, and this anomaly was safely ignored. Like VMWare, this example shows that given a sufficient period of time (in this case, several months) and system calls (billions), pH can capture the normal behavior of complicated programs.

Approximately 5 of the 23 tolerizations during this period were due to experiments involving *inetd* (see Chapter 7); if we remove these actions, we are left with 1.15 tolerizations per day. As Table 6.12 shows, this value is more than double the rate from the other three hosts, but it is still relatively low, and is less than half the rate of the earlier tests on my home computer.

These results show that it is practical to run pH with a small *suspend_execve* threshold, provided that there is a person or daemon process which can evaluate suspended processes. With simple delays, most false positives are transient, and tolerizations are generally used to speed up sluggish programs. Processes delayed by the *suspend_execve* mechanism, however, are effectively killed unless some action is taken. Even if such responses are rare, Murphy's law guarantees that they will happen at the worse possible moment, potentially resulting in a significant loss of functionality or data. Fixing such situations only requires a few mouse clicks with *pHmon*; of course, users first need to be educated about pH before they can perform these actions.

As with so many other security mechanisms, pH can provide better security at the cost of more administration. Yet, because this administration involves evaluating simple, easy-to-understand responses through a simple graphical interface, pH can be maintained by unsophisticated users.

6.10 Normal in Practice

Although the preceding figures and tables are quantitatively accurate descriptions of how pH behaves, they do not fully convey the "feel" of pH. In practice I have found that with the settings in Table 6.10, pH almost always delays programs because of some change in the usage or configuration of the machine. Because I frequently change my usage patterns, I tend to encounter a relatively high rate of false positives; as the badshot dataset shows, other users with more consistent usage patterns have reported many fewer problems.

On lydia, false positives almost always come from me using a program in a new way, either directly or indirectly. For example, I once generated a number of anomalies in response to running a new program, *kdf*. This program is a KDE interface to *df*, a standard utility for obtaining the amount of free space on mounted volumes. Because I frequently use *df* on the command line, pH had classified its profile as normal. *kdf* invokes *df* but uses options that I normally do not use. pH detected this difference and responded with significant delays. Another time pH delayed *cron* after I had commented out an entry in a crontab file, removing a normally-run job. pH detected and reacted to the change in *cron*'s behavior.

Because pH reacts by slowing down abnormally-behaving programs, however, false positives have not created many problems for me in practice. If the program is behaving more slowly than I would like, I merely click on the process in *pHmon* and tell pH to tolerize it. Unless a timeout has somehow been triggered (a rare event, unless the anomaly happened overnight), execution then proceeds normally.

Nevertheless, reactions such as these have changed the way that I interact with my computer. Before choosing a new activity, I now first wonder how pH will react to the change. When my computer hesitates, I automatically ask myself what have I done differently. By detecting and delaying novel program behavior, pH has given

me the sense that my computer doesn't like change. As long as I continue to do the things I have done in the past everything behaves as expected. If I decide to change the way background services run, though, or if I use an old program in a new context, I often expect pH to react. pH often ignores actions that I suspect might set it off; also, after periodically correcting pH's behavior, over time I find that pH reacts less and less often to my direct actions. Even so, with a bit of thought I can generally set it off. My normal usage patterns, however, do not alarm pH at all.

pH has also changed the way I administer my computer. In December 2001, I bought a new printer for lydia. To use this printer I had to install some new software. Installing a new program can cause a few pH anomalies; this software, however, required me to upgrade my C library as well. Because almost every program on a UNIX system is dynamically linked to the C library, this one upgrade caused almost every program on my system to behave anomalously. To make my system usable again, I had to tolerize manually every profile on my system. Although I was unhappy with the need for these tolerizations, they were simple to perform: I rebooted in single-user mode and executed a one-line command.

Yet in contrast to systems like Tripwire [59], pH does not always react to program upgrades. In particular, security-related upgrades normally generate no anomalous program behavior. This observation is partially a by-product of Debian's security policy: instead of upgrading packages when security fixes are incorporated, the Debian project backports security fixes to the older program versions that are part of the stable Debian distribution. Because Debian works hard to ensure that security fixes are made with minimal source code changes, security upgrades are almost guaranteed to not interfere with the stability of a system. This same stability also keeps pH from reacting to security updates that do not significantly change program behavior.

To better understand pH's behavior, the next chapter analyzes pH's anomalies.

Chapter 7

Anomalous Behavior

Once pH has a profile of normal program behavior, it then monitors subsequent program behavior, delaying any unusual system calls. In this chapter, I explore the nature of these anomalies, and examine how they correlate with unusual and dangerous events. The first part discusses what type of changes in program behavior should be detectable by pH by examining the system calls produced by a few simple programs. The next section examines how pH detects and responds to changes in *inetd* behavior and shows that pH can detect some kinds of configuration errors and the usage of normally disabled internal services.

Following this, I test whether pH can detect and respond appropriately to a *fetchmail* buffer overflow, an *su* backdoor, and a kernel ptrace/execve race condition that allows a local user to obtain root access. The attacks are explained and the specific detected behaviors are explored. I then summarize pH's actual or simulated response to several other previously tested intrusions and explain how an intelligent adversary might try to avoid detection by pH. The chapter is concluded with a discussion of the effectiveness of pH's responses.

Since abnormal program behavior can only be defined in terms of normal behav-

```
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char *msg = "Hello World!\n";

        write(1, msg, strlen(msg));

        return 0;
}
```

Figure 7.1: *hello.c*: A simple "hello world" program, written to use a minimal amount of library code for output.

ior, all of the anomalies reported in this chapter are relative to specific lookahead-pair system call profiles. Some of these normal profiles reflect program usage on one or more computers ("real" normal profiles); however, for programs that were not as frequently used the profiles reflect deliberate tests of program functionality ("synthetic" normal profiles). Chapter 6 discussed some of the trade-offs of real and synthetic normal profiles.

Except where otherwise noted, pH's parameters were set to the default values in Table 5.2.

7.1 What Changes Are Detectable?

By detecting changes in the pattern of system calls, pH is able to observe changes in the flow of control of a program. Some changes in flow will not be detectable if they produce previously seen lookahead call patterns; however, novel lookahead pairs are proof that a previously unseen flow of control is being observed. Although a previously unseen execution path may be perfectly safe, pH assumes that such a

```
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char *msg = "Hello World!\n";

        if (argc > 1) {
                execl("/bin/ls", "ls", NULL);
        } else {
                write(1, msg, strlen(msg));
        }

        return 0;
}
```

Figure 7.2: *hello2.c*: A second simple "hello world" program, but which executes the *ls* command when given any command line arguments.

path is potentially dangerous and so merits a response.

To better see what this means in practice it is helpful to look through a few simple examples. The programs listed in Figures 7.1, 7.2, and 7.3 all print "Hello, World!" when run without any command-line arguments. Without arguments these programs also produce exactly the same trace of 22 system calls, which are shown in detail in Figure 7.4. This figure shows the output of strace, and except for the initial execve, is identical to the trace as seen by pH. Of these calls, only the write system call is actually produced by the main part of the program: the calls before this point are made by the dynamic linker and the C library pre-main initialization routines, while the final exit terminates the program after the main function returns.

Because there are no branches in the main function of *hello.c*, the write system call will always be executed; however, the context of this write may change if the system's configuration changes. For example, the fstat64 call is implemented in Linux

```
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char *msg1 = "Hello World!\n";
        char *msg2 = "Goodbye World!\n";

        if (argc > 1) {
                write(1, msg2, strlen(msg2));
        } else {
                write(1, msg1, strlen(msg1));
        }

        return 0;
}
```

Figure 7.3: *hello3.c*: A third simple "hello world" program which also says goodbye when given a command-line argument.

2.4 kernels: when the C library is run on such a system it detects fstat64's presence and changes its initialization appropriately. Except for such initialization changes, we would expect pH to generate a profile containing 156 lookahead pairs and we would never expect to see any anomalies relative to this profile.

If the program *hello2.c* is run with no arguments, it produces the same trace of system calls as *hello.c* and generates an identical profile. If we give *hello2.c* an argument, we instead see the write replaced with an execve call (see Figure 7.5). This substitution generates one anomalous system call (the execve) if the profile is marked as normal. If we are still training, this change generates eight new lookahead pairs, one for each system call preceding the execve. After the execve, control passes to the program *ls* and its profile, and so no further novel behavior is detected. In this fashion, pH can detect the novel code path triggered by the presence of a command-line argument by detecting the anomalous execve call. In addition, if the

suspend_execve variable was set to 1 and the profile for *hello2.c* was marked as normal, the anomalous execve would automatically trigger a two-day delay before *ls* was run.

The execution of *hello3.c* is similarly perturbed by the presence of any command-line arguments; its profile, however, is not perturbed by this change, and if the profile is marked as normal, it produces no anomalies. pH cannot detect this change because the alternative code path invokes exactly the same system call (write) as the "normal" code path, and pH does not examine system call arguments (see Figure 7.6).

These simple examples show how pH detects and responds to significant changes in a program's flow of control. In these examples the perturbations were caused by changes in command line arguments. With more complicated programs, anomalies can be caused by similar changes, e.g., by a user trying out using `--help` as an option. Although this action might appear innocuous, such anomalies do not occur frequently in practice. Thus, an otherwise benign anomaly can indicate the presence of an unauthorized user who was unfamiliar with the system. To be sure, this would be weak, indirect evidence of a security violation; as Section 7.3 shows, pH can also detect security violations directly.

```
execve("./hello", ["hello"], [/* 36 vars */]) = 0
uname({sys="Linux", node="lydia", ...}) = 0
brk(0) = 0x804957c
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
open("/etc/ld.so.preload", O_RDONLY) = -1
   ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, 0xbfffee44) = -1
   ENOSYS (Function not implemented)
fstat(3, {st_mode=S_IFREG|0644, st_size=67285, ...}) = 0
old_mmap(NULL, 67285, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40016000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0X\324\1"...,
   1024) = 1024
fstat(3, {st_mode=S_IFREG|0755, st_size=1124904, ...}) = 0
old_mmap(NULL, 1141380, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0)
   = 0x40027000
mprotect(0x40134000, 39556, PROT_NONE) = 0
old_mmap(0x40134000, 24576, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED, 3, 0x10c000) = 0x40134000
old_mmap(0x4013a000, 14980, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4013a000
close(3) = 0
munmap(0x40016000, 67285) = 0
getpid() = 11318
write(1, "Hello World!\n", 13) = 13
_exit(0) = ?
```

Figure 7.4: The 22 system calls emitted by *hello.c*, as reported by *strace*.

```
execve("./hello2", ["hello2", "foo"], [/* 36 vars */]) = 0
uname({sys="Linux", node="lydia", ...}) = 0
brk(0) = 0x80495ec
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
open("/etc/ld.so.preload", O_RDONLY) = -1
   ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, 0xbfffee34) = -1
   ENOSYS (Function not implemented)
fstat(3, {st_mode=S_IFREG|0644, st_size=67285, ...}) = 0
old_mmap(NULL, 67285, PROT_READ, MAP_PRIVATE, 3, 0)
   = 0x40016000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0X\324\1"...,
   1024) = 1024
fstat(3, {st_mode=S_IFREG|0755, st_size=1124904, ...}) = 0
old_mmap(NULL, 1141380, PROT_READ|PROT_EXEC,
   MAP_PRIVATE, 3, 0) = 0x40027000
mprotect(0x40134000, 39556, PROT_NONE)  = 0
old_mmap(0x40134000, 24576, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED, 3, 0x10c000) = 0x40134000
old_mmap(0x4013a000, 14980, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4013a000
close(3) = 0
munmap(0x40016000, 67285) = 0
getpid() = 11331
execve("/bin/ls", ["ls"], [/* 36 vars */]) = 0
...
[System calls of /bin/ls]
...
_exit(0) = ?
```

Figure 7.5: The system calls emitted by *hello2.c* when given a command line argument, as reported by *strace*. Note how the boldface region is different from *hello.c* (Figure 7.4).

```
execve("./hello3", ["hello3", "foo"], [/* 36 vars */]) = 0
uname({sys="Linux", node="lydia", ...}) = 0
brk(0) = 0x80495bc
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
open("/etc/ld.so.preload", O_RDONLY) = -1
   ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, 0xbfffee34) = -1
   ENOSYS (Function not implemented)
fstat(3, {st_mode=S_IFREG|0644, st_size=67285, ...}) = 0
old_mmap(NULL, 67285, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40016000
close(3) = 0
open("/lib/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0X\324\1"...,
   1024) = 1024
fstat(3, {st_mode=S_IFREG|0755, st_size=1124904, ...}) = 0
old_mmap(NULL, 1141380, PROT_READ|PROT_EXEC,
   MAP_PRIVATE, 3, 0) = 0x40027000
mprotect(0x40134000, 39556, PROT_NONE)  = 0
old_mmap(0x40134000, 24576, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED, 3, 0x10c000) = 0x40134000
old_mmap(0x4013a000, 14980, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4013a000
close(3) = 0
munmap(0x40016000, 67285) = 0
getpid() = 11333
write(1, "Goodbye World!\n", 15) = 15
_exit(0) = ?
```

Figure 7.6: The system calls emitted by *hello3.c* when given a command line argument, as reported by *strace*. Note how the arguments to the last write call (in boldface) are different from those of *hello.c* (Figure 7.4).

Program	/usr/sbin/inetd
Version	Debian netkit-inetd 0.10–8
Window Size	9
Total Calls	3563667
Days of Training	37
Lookahead Pairs	1467
Novel Sequences	385

Table 7.1: Details of the *inetd* normal profile used for perturbation experiments. Note that the number of novel sequences refers to the number of sequences that contained new lookahead pairs.

7.2 Inetd Perturbations

Sometimes pH detects anomalies because of benign changes in usage patterns; other times it detects anomalies because of significant changes in program functionality. These changes can be of interest to a system administrator even when they do not represent a security threat because they can indicate a configuration error or the presence of an undesirable service. In this section I explore how pH responds to such changes in the behavior of *inetd*.

The *inetd* program is a kind of "super-server" daemon that runs on most UNIX machines to provide access to a variety of services. It has a configuration file called `inetd.conf` which associates TCP and UDP ports with different built-in services or external executables. When started, it binds the ports listed in this file, and upon connection to any of these ports it responds by either running an internal routine (for built-in services) or by spawning an external program. Because of the overhead of invoking a new process on every connection, *inetd* generally is not used for high-volume services; instead, it is used for needed services that are not run frequently enough to justify having their own dedicated daemon.

Although the `inetd.conf` file is the source of much of *inetd*'s flexibility, it is also

Count	LFC	Delay (s)	Call #	Call Name
302	1	0.02	175	rt_sigprocmask
303	2	0.04	13	time
304	3	0.08	5	open
305	4	0.16	108	fstat
306	5	0.32	90	mmap
307	6	0.64	3	read
309	7	1.28	91	munmap
310	8	2.56	4	write
311	9	5.12	6	close
312	10	10.24	142	newselect

Table 7.2: System call anomalies for the *inetd* daytime service. "Count" refers to the position in the trace of the anomalous *inetd* process.

the source of many problems. Secure systems typically disable all of *inetd*'s internal services since they provide at best non-critical services, many of which can be used for denial-of-service attacks; however, the default install of some systems still leaves these services enabled. Also, typographical errors in the configuration file can accidentally disable desired services. In the rest of this section I examine the impact of these types of `inetd.conf` changes by examining four perturbations: an enabled daytime service, an enabled chargen service, a misspelled filename in the finger service, and a misspelling of the finger service designation. The data is presented relative to a normal profile of *inetd* behavior from August 10th through September 17th, 2001 generated on my home computer, lydia. This profile is summarized in Table 7.1. Also, where pH's data was insufficient to reconstruct the behavior of the program, *strace* was used to obtain arguments to the system calls.

```
 for (each requested service) {
      identify service
      pid = 0
      if (dofork) {
           do the fork + many system calls
           pid == 0 in child
      }
      ...
      clear blocked signals (sigprocmask(empty mask))
      if (pid == 0) {
           if (dofork) {
                setup uid, gid, etc.
                duplicate standard file descriptors
closelog()
           }
           if (built-in service) {
                run service
           } else {
                execv external program
           }
      }
 }
```

Figure 7.7: Pseudo-code of the *inetd* main loop.

7.2.1 Daytime

The TCP daytime service [88] is an extremely simple network service where a remote machine may connect to port 13 on a server to receive a human-readable text string describing the current time. Although most Internet systems are capable of supporting this service, it is generally disabled because it provides more information about a server than is strictly necessary, and because any open port can represent a potential vulnerability.

To see how pH would react to the enabling of daytime, the `inetd.conf` file on lydia was changed to enable the service. Restarting *inetd* produced no anomalous

system calls; a telnet to port 13, though, produced 10 anomalous system calls with a maximum LFC of 10. With $delay\_factor$ set to 1, *inetd* was explicitly delayed for 21.1 seconds, which resulted in an overall delay of less than 30 seconds. The anomalies and delays are detailed in Table 7.2. Because the last anomalous system call was a select, *inetd* still had an LFC of 10 when the daytime access completed. Thus, the next 117 system calls (for subsequent requests) would each be delayed for 10.24 seconds[1].

The daytime service generated these anomalies because of three factors: the service didn't require another process to be spawned, another executable didn't need to be run, and the built-in daytime() service function executes the time system call which is not invoked for any external service. To see this more clearly consider the pseudo-code of *inetd*'s main loop in Figure 7.7. For a normal external, nonblocking service, *inetd* follows the branches corresponding to dofork being true and of the request being for an external service. Neither of these conditions are true for daytime. Further, the daytime() function invokes the time system call, a call which isn't used by the external service code path.

Thus, pH does detect that *inetd* is behaving in an unusual manner; however, its response causes only a relatively minor delay in service, and even worse, delays on subsequent service requests. Nevertheless, the anomalies and the subsequent delays can alert an administrator that an unneeded service is enabled and being used.

7.2.2 Chargen

The TCP chargen service [87] is a simple network service designed to help test the reliability and bandwidth of a connection. Upon connection to port 19, the server

[1]To prevent delay of the main *inetd* daemon from causing a denial of service, we can borrow a trick from the software rejuvenation field [20, 111] and use a "watchdog" daemon to restart *inetd* whenever it is substantially delayed.

Count	LFC	Delay (s)	Call #	Call Name
35	1	0.02	63	dup2
36	2	0.04	6	close
37	3	0.08	63	dup2
38	4	0.16	63	dup2
40	5	0.32	6	close
1059	1	0.02	102	socketcall
1060	2	0.04	4	write
1061	3	0.08	4	write
1062	4	0.16	4	write
1063	5	0.32	4	write
1064	6	0.64	4	write
1065	7	1.28	4	write
1066	8	2.56	4	write
1067	9	5.12	4	write
1068	10	10.24	4	write
1069	11	20.48	4	write
1070	12	40.96	4	write
1071	13	81.92	4	write
1072	14	163.84	4	write
1073	15	327.68	4	write
1074	16	655.36	4	write
1075	17	1310.72	1	exit

Table 7.3: System call anomalies for the *inetd* chargen service. "Count" refers to the position in the trace of the anomalous *inetd* process.

sends a full-speed continuous stream of repetitive ASCII characters until the client terminates the connection. Because the service obeys TCP congestion control, in theory it should not consume an excessive amount of bandwidth; in practice, such throttling is not sufficient to prevent abuse. A forged connection (created through spoofed IP packets) can cause data to be sent to an arbitrary target machine. Several chargen streams converging on a given network can constitute an effective denial-of-service attack; thus, most current machines have this service disabled.

As with daytime, enabling chargen caused no anomalies during *inetd*'s startup.

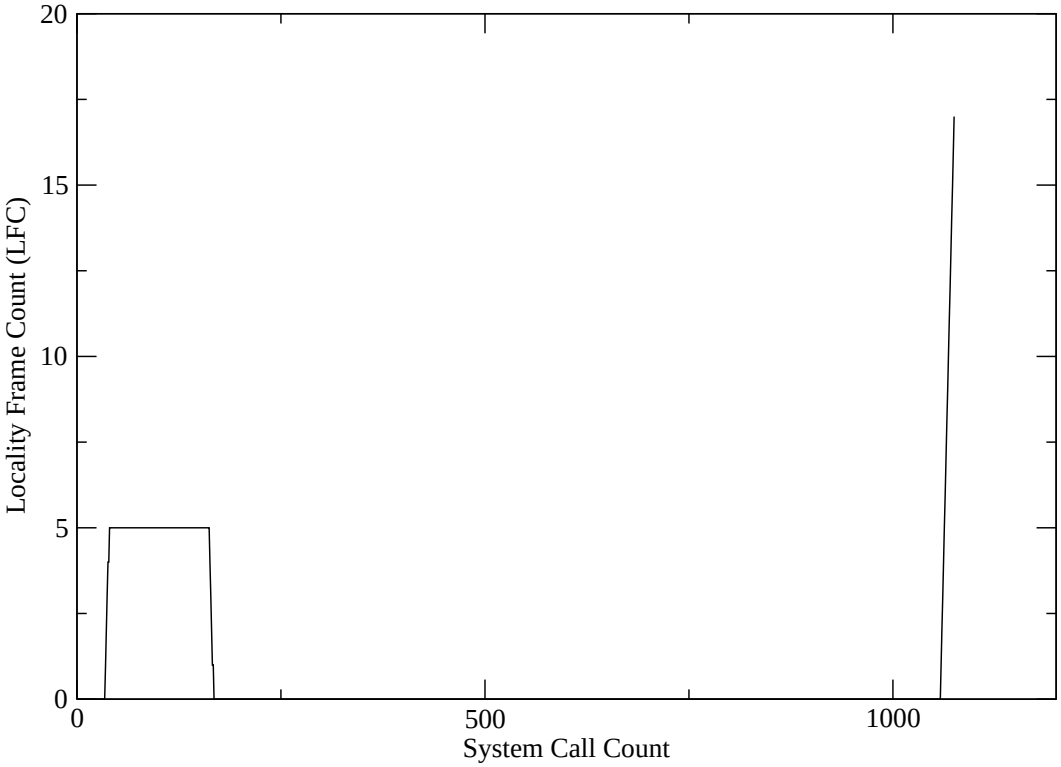

Figure 7.8: The locality frame count of the *inetd* chargen anomalies, as seen with delays enabled. Note that each system call is delayed for $2^{LFC}/100$ seconds.

A telnet to port 19, however, quickly produced a flood of anomalies. With all delays disabled, pH recorded over 13,000 anomalies within the first minute of the connection. With $delay\_factor$ set to 1, only 22 anomalies were generated during the first 23 minutes of the connection, achieving a maximum LFC of 17 (see Table 7.3 and Figure 7.8). Only 13 lines of output were produced in contrast to a few megabytes with no response. Thus, delays are sufficient to stop the use of this unneeded service. Because the maximum LFC for these anomalies exceeded the $tolerize\_limit$ of 12, pH reset *inetd*'s training profile, preventing pH from learning the use of chargen as normal *inetd* behavior.

Like the daytime service, chargen is built in to *inetd*; the anomalies generated by chargen, however, are rather different from those for daytime. There are two sets

of anomalies. The first set corresponds to the second dofork code path shown in Figure 7.7, the part where the uid and gid of the child process is changed. None of the other regularly used external programs on lydia run as root, so system calls are made to change the process's uid and gid appropriately. In contrast, chargen does run as root. Since *inetd* is already running as root, *inetd* never makes any system calls to change its uid or gid; instead, it skips ahead and duplicates the standard file descriptors, producing the "dup2, close, dup2, dup2" calls in Table 7.3. The second close comes from a call to closelog() function, which closes the process's connection to syslog.

The second set of anomalies correspond to the chargen_stream() function — a function that is normally never invoked on lydia. Here, *inetd* sets the name of the process after finding out the name of the socket's peer (socketcall) and then repeatedly writes a set of bytes to the socket (multiple write's). When one of these writes returns an error, the process terminates with an exit system call.

7.2.3 Unknown External Program

Because the `inetd.conf` file is often edited by hand, there is a significant risk that a typographic error can invalidate part of the file. One error that can go unnoticed is that of an incorrectly spelled filename. pH can detect such problems if they arise from recent configuration changes and can help alert a system administrator as to the problem.

For an example of this type of error, consider the following configuration line:

```
finger stream tcp nowait nobody /usr/sbin/tcpd /usr/sbin/in.fingerd
```

This line is an entry in `/etc/inetd.conf` for finger [122], a service that allows one to find out who is logged in to a given machine. Because the Internet is a much

Count	LFC	Delay (s)	Call #	Call Name
1074	1	0.02	45	brk
1075	2	0.04	13	time
1076	3	0.08	5	open
1077	4	0.16	108	fstat
1078	5	0.32	90	mmap
1079	6	0.64	3	read
1080	7	1.28	6	close
1081	8	2.56	91	munmap
1089	9	5.12	1	exit

Table 7.4: System call anomalies for *inetd* with a configuration entry for a non-existent executable. Note that these delays are for the spawned child process and not for the parent server process.

less trusting place now than it used to be, it is common for finger to be disabled or restricted to certain hosts. The */usr/sbin/tcpd* file is part of the TCP Wrappers package [113], which is used to restrict access on the basis of DNS names or IP addresses. So, when *inetd* receives a finger request, it spawns a process which then runs *tcpd*. This program tests to see if the access should be allowed, and if so it invokes *in.fingerd*. Consider what would happen if we were to replace */usr/sbin/tcpd* with */usr/sbin/tcp*. The two filenames look very similar, but the latter doesn't exist.

It turns out that pH detects no anomalies when *inetd* starts with this configuration error; however, once a finger request is received, pH detects 9 anomalies and delays the spawned child process for approximately 10 seconds before it exits. The behavior of the child process is detailed in Table 7.4. Note that the main server *inetd* process is unaffected.

These anomalies come straight from this fragment of *inetd*'s main() function:

```
execv(sep->se_server, sep->se_argv);
if (sep->se_socktype != SOCK_STREAM)
```

```
        recv(0, buf, sizeof (buf), 0);
    syslog(LOG_ERR, "execv %s: %m", sep->se_server);
_exit(1);
```

If the execv() call succeeds, the appropriate external program replaces *inetd* in the spawned child process. In this case, however, the call fails because the specified file does not exist. The anomalous system calls all come from the syslog() and _exit() calls. The syslog() function obtains some additional memory (brk), gets the time, reads `/etc/localtime` to find out the current time zone (open through the munmap), connects to the syslog socket (no anomalies), and then returns. The final anomaly is the _exit() call, which merely calls the exit system call to terminate the process.

With this sort of error, pH's delay response does not make much of a difference. The service is incorrectly specified, so there is no additional denial of service. The main daemon is unaffected, therefore connections for other services continue to be serviced. The anomalies and delays, however, do serve to alert an administrator that there is a problem with *inetd*'s configuration.

7.2.4 Unknown Service

In the last section I examined how a typographic error in a filename could cause anomalies. In this section, I examine a similar error: a misspelling of the service name. At the beginning of each line is an ASCII string or a number representing the port upon which *inetd* should listen for requests. If this service is a TCP or UDP service, the ASCII name to number mapping is specified in the `/etc/services` file.

To test how pH would react to a service name not listed in `/etc/services`, the letters of the finger service were transposed to "figner" and *inetd* was restarted. As shown in Table 7.5, during *inetd*'s initialization pH detected 15 anomalies and delayed

Count	LFC	Delay (s)	Call #	Call Name
120	1	0.02	3	read
121	2	0.04	6	close
122	3	0.08	91	munmap
134	4	0.16	3	read
135	5	0.32	6	close
136	6	0.64	91	munmap
137	7	1.28	45	brk
138	8	2.56	13	time
139	9	5.12	5	open
152	10	10.24	5	open
153	11	20.48	55	fcntl
155	12	40.96	108	fstat
156	13	81.92	90	mmap
157	14	163.84	3	read
158	15	327.68	3	read

Table 7.5: System call anomalies for *inetd* with a configuration entry for a non-existent service.

it explicitly for almost 11 minutes. The next 113 system calls (which would be made by subsequent requests) would each be delayed for over five minutes, meaning that we wouldn't expect normal responses times for at least 10 hours.

Unlike the three previous perturbations, these anomalies were generated by *inetd*'s configuration file parsing routine. In the config() routine, *inetd* reads through each line of `inetd.conf` and sets up the ports and data structures needed for each service. The fact that the "figner" service does not exist causes this function and the routines it calls to behave differently than normal.

The anomalies in Table 7.5 split into three groups: by count, they are 120–122, 134–139, and 152–158. The middle group is the easiest to explain. It is generated by the following bit of code in config():

```
u_short port = htons(getservbynum(sep->se_service));
```

```
if (!port) {
        struct servent *sp;
        sp = getservbyname(sep->se_service,
                                sep->se_proto);
        if (sp == 0) {
                syslog(LOG_ERR,
                    "%s/%s: unknown service",
                    sep->se_service, sep->se_proto);
                continue;
        }
        port = sp->s_port;
}
```

The "figner" service is not a number, so the `(!port)` branch is taken. The code then calls getservbyname(), which consults `/etc/services` for the right port number. Since there is no appropriate entry in this file, the `(sp == 0)` branch is taken, and an error is logged. The loop then proceeds to check the next service entry.

System calls 137–139, the brk, time, open sequence, correspond to the call to syslog(). Calls 134–136, though, are generated by the getservbyname() routine. It turns out that other invocations of this routine to look up existing services require at most four reads. Because figner is not in `/etc/services`, getservbyname() has to read the entire file, which requires *five* reads. This extra-long sequence of reads, along with the fact that this routine mmap's the file to speed access, together generate these three anomalies. Similarly, 120–122 come from a previous invocation of getservbyname() within the same loop iteration in a "silly" code fragment (according to the source comments) which is designed to detect multiple entries of the same service that use different aliases. The third set, 152–158, also corresponds to the "silly" code

getservbyname(), but on the next loop iteration. The `continue` statement causes it to be executed prematurely; normally a valid service would be initialized before returning to the top of the loop.

From the above description, it is apparent that pH does detect novel code paths, although sometimes its view of novelty doesn't correspond to one's intuitive understanding of the code. Because pH sees things at an extremely low level and observes actual behavior, it often uncovers patterns not apparent at the source level. Also, pH's response to an unknown service potentially does result in a denial of service; since this sort of anomaly would only arise after a change to *inetd*'s configuration file, though, such a response is more likely to alert a system administrator to a potential problem rather than actually cause a true loss of service.

7.2.5 Summary

These four examples show that pH can detect interesting errors in *inetd*'s configuration by detecting novel code paths. In three of the four cases, pH's delay response merely serves as an alert to an administrator that things aren't quite as they should be; however, in the case of the chargen service, pH's delays prevents a misconfigured service from being used as a tool for distributed denial-of-service attacks.

Also, the regularities that pH uses to distinguish between normal and abnormal behavior can be difficult to determine from the source of a program. Almost any program (even in a low-level language like C) calls library functions, each of which can make a significant number of system calls. As was shown in the last example, detected anomalies may have their roots in changes in the behavior of these library functions.

Although such low-level regularities (and irregularities) may make diagnosis difficult, they also serve as examples of the complexity and inefficiency of common pro-

grams. For example, the getservbyname() function reads the entire contents of the `/etc/services` file every time it is called. *inetd* calls getservbyname() twice for each configured service; thus, if *inetd* has ten services configured, it will indirectly read the `/etc/services` file twenty times. In contrast, if *inetd* read in `/etc/services` manually and stored its contents in a hash table, it could have accessed the file only once. Since getservbyname() is only called during *inetd*'s initialization, these redundant (though individually efficient) file accesses do not impose a significant performance penalty; nevertheless, such behaviors are a small example of how the increasing modularity of current systems leads to numerous inefficiencies.

7.3 Intrusions

In the last part, pH was shown to be able to detect configuration errors and normally unused services. In this section, I will address how pH is able to detect and respond to security violations. Because most attacks involve normally unused code paths, their dangerous behavior is detectable; further, a delay-oriented response can be very effective in thwarting these attacks.

In our past work we showed that several different kinds of attacks can be detected through system-call monitoring [43, 49]; what we haven't addressed is why pH is able to detect them. Therefore in this section I will focus on trying to understand what sorts of behaviors pH is able to detect along with behaviors it will miss. By showing how three attacks exploiting three distinct vulnerabilities can be both detected and stopped, it should become apparent that pH can flexibly deal with a wide variety of security violations.

```
static int pop3_getsizes(int sock, int count, int *sizes)
/* capture the sizes of all messages */
{
    int ok;

    if ((ok = gen_transact(sock, "LIST")) != 0)
        return(ok);
    else
    {
        char buf [POPBUFSIZE+1];

        while ((ok = gen_recv(sock, buf, sizeof(buf))) == 0)
        {
            int num, size;

            if (DOTLINE(buf))
                break;
            else if (sscanf(buf, "%d %d", &num, &size) == 2)
                sizes[num - 1] = size;   /* OVERFLOW */
        }

        return(ok);
    }
}
```

Figure 7.9: The function pop3_getsizes() from *fetchmail* 5.8.11. Note how the line marked "OVERFLOW" assigns values to the sizes array without doing any bounds checking.

Program	/usr/bin/fetchmail
Version	fetchmail 5.8.11
Window Size	9
Total Calls	4461219 (4464889)
Lookahead Pairs	2597 (2683)
Novel Sequences	725 (767)

Table 7.6: Details of the *fetchmail* normal profile used for the buffer overflow experiments. The values in parentheses refer to the augmented profile to which interactive fetchmail behavior was added.

7.3.1 A Buffer Overflow

A surprisingly common type of vulnerability is the buffer overflow. Although in practice they can be both complex and subtle, buffer overflows are simple in concept. They all involve a program using a fixed amount of storage to hold data that is influenced by external input. If the program does not properly ensure that the data fits within the allocated space, adjacent memory locations can be overwritten, potentially causing foreign data and code to be inserted and even executed. Such overflows may overwrite variables, pointers, or stack-resident function return addresses, and in doing so they can influence program behavior in arbitrary ways, limited only by the imagination and determination of the attacker.

Because buffer overflows are a well-known and widespread problem, numerous solutions have been proposed (see Chapter 2). pH is also able to detect and stop buffer overflow attacks; however, instead of detecting dangerous stack modifications [32] or preventing stack-based code from being executable [37], pH detects buffer overflows by recognizing the unusual, malicious code that is inserted into the victim program. To see why pH is effective, consider the following attack against *fetchmail*.

fetchmail is a utility that allows one to retrieve email from a variety of remote servers using POP3, IMAP, and other protocols. The retrieved email is delivered

using the local mail transport program, making it appear to have been delivered directly to the user's workstation. Standard UNIX email clients can then access the user's email spool file normally without having to know how to access the remote servers.

In August 2001, Sanfilippo [4] publicized a buffer overflow vulnerability in *fetchmail* versions prior to 5.8.17. The vulnerable code is in the pop3_getsizes() and imap_getsizes() functions of *fetchmail*. The dangerous code is extremely similar in both functions, and so for this discussion I will focus on pop3_getsizes().

The error in this function comes from the following two lines (see Figure 7.9):

```
else if (sscanf(buf, "%d %d", &num, &size) == 2)
    sizes[num - 1] = size;   /* OVERFLOW */
```

These lines read message lengths from the POP3 server and place them in the `sizes` array. The code is compact and elegant; unfortunately, it does no sanity checking on the value of `num`. If the server is malicious and returns message indices which are outside the range previously given, it can cause fetchmail to write 4-byte values to arbitrary stack memory locations.

Sanfilippo's advisory included a sample exploit script which behaves as a malicious POP3 server and causes fetchmail to execute an *ls* of *fetchmail*'s working directory. After adjusting a constant in the exploit's code which specified the memory location of the `sizes` array, I was able to successfully exploit a fetchmail 5.8.11 executable[2].

When the sample exploit was run while pH was monitoring with the profile in

[2]Although it is possible to create more robust buffer overflow exploits, most available attack scripts are extremely brittle, and are specialized to the memory layout of specific binaries. Thus, adjusting memory constants is a normal part of making buffer overflow attacks work in practice.

Table 7.6, pH detected 19 anomalies with a maximum LFC of 15. These anomalies caused pH stop the attack by delaying fetchmail for several hours, even with *suspend_execve* set to 0. This result is not entirely fair, though, in that almost all of these anomalies occur because the sample exploit requires *fetchmail* to be invoked in interactive mode. My normal usage of *fetchmail* is in daemon mode in which it periodically retrieves mail in the background. To make the test more challenging, I then augmenting the normal profile with a few interactive invocations of *fetchmail*. With this expanded normal profile, pH reported one anomalous system call, an execve present in the injected code. Since this anomaly was an execve, pH was able to stop the attack, but only with *suspend_execve* set to 1.

Sometimes pH detects buffer overflows because the exploit's memory corruption causes unusual program behavior before the inserted code can take control. In situations such as this, though, the foreign code takes control before any unusual system calls get made, and so pH must detect and respond to calls made by the inserted code. As explained in the next section, it is at least sometimes possible for the foreign code to be modified to make system call sequences that look normal; current attacks, though, take no such precautions.

7.3.2 Trojan Code

Most programs contain features that are normally unused. Many of these features are safe; others, though, can have severe security consequences. Some of the most dangerous "features" are provided by trojan code, or code designed to circumvent existing security mechanisms. For example, it is common for intruders to install "root kits" containing several compromised system binaries once they have gained access to a system; once this has been completed, they effectively control that machine even if an administrator removes the original vulnerability. [D: a few random cites?] On such

Program	/usr/su
Version	Debian login 20000902-6.1
Window Size	9
Total Calls	18021
Lookahead Pairs	1377
Novel Sequences	318

Table 7.7: Details of the *su* normal profile used for the trojan back door and kernel race experiments. Note that the number of novel sequences refers to the number of sequences which contained new lookahead pairs.

a modified system, daemons such as *ssh* will have "back doors" which give a remote user full system access through a password not contained in the system password file. Even worse, monitoring programs such as *ps* and *netstat* often are replaced with versions that hide the presence of the intruder by masking their processes and network connections. Once a root kit has been installed, generally the only safe course is to do a complete re-install.

Another form of trojan code that can be much more insidious is a back door that is built-in to a program. Perhaps the most famous example of this problem was with an early version of *sendmail*. The vulnerability was extremely simple: a remote user could connect to the SMTP port and type "WIZ". Instantly, that user was given access to a root shell. While such a feature might be useful for debugging, it was also a staggering security vulnerability.

By detecting the novel code paths produced by a trojan program, pH can detect and interfere with the running of these programs. As an example, I added a simple back door to the *su* program. The modification consisted of the following code fragment which was inserted at the beginning of main():

```
if ((argc > 1) && (strcmp(argv[1], "--opensesame") == 0)) {
        char *args[2];
```

```
        char *shell = "/bin/sh";
        args[0] = shell;
        args[1] = NULL;
        execv(shell, args);
}
```

This code allows a user to type `su --opensesame` to gain instant access to a root prompt, without requiring a password.

To see how well pH could detect this back door, it was tested against the normal profile summarized in Table 7.7 (p. 146). This profile was originally generated during the lydia 22-day test period but was later extended as false positives were seen. The profile used for these experiments was used for a week without experiencing any additional false positives.

When the trojan binary was used to run `su -` (the normal usage of *su* on lydia), it generated no anomalies. When the back door was invoked, though, there was one anomaly generated: an execve, which was the 68th system call made by the process. With $delay\_factor$ set to 1, pH delayed the execve and the first 127 system calls of the spawned shell each for 0.02 seconds. This 2.56 second delay was barely noticeable. With *suspend_execve* set to 1, the process was delayed for two days before the execve would complete, effectively preventing the backdoor from being used.

7.3.3 A Kernel Vulnerability

One class of vulnerabilities that are particularly difficult to defend against are kernel implementation errors. Since user-space programs depend upon the kernel for security and integrity services such as memory protection, user-based access control,

Count	LFC	Delay (s)	Call #	Call Name
5	1	0.02	23	setuid
6	2	0.04	46	setgid
7	3	0.08	11	execve

Table 7.8: System call anomalies for *su* produced when the kernel ptrace/execve exploit was run. Note that the execve was suspended for two days after being delayed for 0.08 seconds.

and file permissions, an error in the kernel can lead to a breach in security. Because pH observes program behavior, though, it is capable of detecting and responding to security-related kernel errors.

As an example, consider a subtle vulnerability [92] that affects Linux 2.2 kernels prior to 2.2.19. The vulnerability is a race condition in the execve and ptrace implementations that allows local root access to be obtained. More specifically, if process A is ptrace-ing process B (e.g., A is a debugger, and B is being debugged), and B does an execve of a setuid program, it is possible for process A to control B after the execve. Then, through the ptrace mechanism, process A can make B run arbitrary code. Since the setuid mechanism causes the exec'd program to run with special privileges, process A's modifications to process B would now also run with those special privileges — ones which A did not have previously. In practice, this vulnerability allows a local user to obtain a root shell using almost any setuid-root binary on the system.

To see if pH could detect the exploitation of this hole, I ran Purczynski's sample exploit [92]. The targeted setuid root program I used was *su*, and I used the profile summarized in Table 7.7. Because 2.2.19 is not vulnerable, I back-ported pH-0.18 to a 2.2.18 kernel patched with the 2.4 IDE code and reiserfs. This kernel was only used to test this vulnerability.

The anomalies produced by this exploit are listed in Table 7.8. There are just three anomalies, and they directly correspond to the three system calls present in the inserted code. The exponential delay causes a barely noticeable slowdown of a tenth of a second with $delay\_factor$ set to 1; the *suspend_execve* mechanism, though, detects the anomalous execve and suspends the process for two days, stopping the attack.

7.4 Other Attacks

Having looked at a few attacks in detail, it is reasonable to ask whether these specific results are typical. We have published several papers which report how several attacks can be detected through the analysis of system calls [43, 49, 44, 116, 106]. The rest of this section reviews five past experiments that I performed and addresses how pH performed or should have performed in these tests.

7.4.1 Earlier pH Experiments

In the original pH paper [106], we presented results on three security exploits: an *sshd* (Secure Shell daemon) backdoor [110], an *sshd* buffer overflow [5], and a Linux kernel capabilities flaw that could be exploited using a privileged program such as *sendmail* [91]. This section summarizes these results and discusses how the current version of pH would respond to the same vulnerabilities.

First, it should be noted that these experiments were performed with pH 0.1 instead of pH 0.18. One difference is that this version uses *abort_execve* instead of *suspend_execve*; thus, instead of delaying anomalous execve requests, it causes them to fail. Another difference is that in this older version, the locality frame is not updated if a process's profile is not normal, even though its system calls are delayed

in proportion to its LFC. This difference can cause a newly loaded program to inherit a perpetual delay — something that never happens with the current pH.

These experiments were also performed using different parameter settings: it looked at length 6 sequences (instead of 9) and a $delay\_factor$ of 4 (instead of 1). The locality frame size (128) and $tolerize\_limit$ (12), though, were the same. The net result of these differences is that with the current code and parameter settings, pH would detect more anomalies but would delay each of them for 1/4 of the time.

sshd backdoor

The *sshd* backdoor [110] is a source-code patch for the commercial Secure Shell package (version 1.2.27) that adds a compiled-in password to *sshd*. Secure Shell [97] is a service that allows users to remotely connect to a UNIX machine using an encrypted communications channel. When a remote user connects but uses the built-in password, *sshd* bypasses its normal authentication and logging routines and instead immediately provides superuser access. This modified binary can be installed on a previously compromised machine, providing the attacker with an easy means for regaining access.

pH's responses to this backdoor were tested relative to a synthetic normal profile of the modified *sshd* daemon. In these tests, use of the backdoor generated 5 anomalies: 2 (LFC 2) in the primary *sshd* process, and 3 (LFC 3) in the child process spawned to service the attack connection. This number of anomalies was not sufficient, in itself, to stop the use of the backdoor; with $abort\_execve$ set to 1, though, the child process was unable to perform an execve, preventing the remote user from obtaining root access. pH 0.18 would react similarly to these attacks, except that it would probably detect a few more anomalies because of the longer default lookahead pair window size.

Note that because the main server process experiences a maximum LFC of 2, child processes created to service future connections will also have a maximum LFC of 2. If *suspend_execve* is set to 1, this will cause these post-attack connections to be delayed for two days until *sshd* is either tolerized or restarted.

sshd buffer overflow

The *sshd* buffer overflow attack [5] exploits a buffer overflow in the RSAREF2 library which optionally can be used by *sshd*. To test this attack, *sshd* version 1.2.27 was built and linked to the RSAREF2 library, and a synthetic normal profile was generated for this binary[3]. The attack program, a modified *ssh* client, caused the primary *sshd* process to execute 4 anomalous system calls (LFC 4).

Simple delays were not sufficient to stop this attack; however, with pH 0.1, the exec'd *bash* shell inherited an LFC of 4 that caused every bash system call to be delayed for 0.64 seconds (with a $delay\_factor$ of 4). Setting $abort\_execve$ to 1, though, caused the attack to fail.

With pH 0.18 and a $delay\_factor$ of 1, pH would only delay the first 125 system calls of the exec'd *bash* shell for 0.16 seconds; shortly after that, the shell's system calls would have no delay at all. With $suspend\_execve$ set to 1, though, the current pH version would stop the attack before the *bash* shell were run, and instead the primary *sshd* daemon would be delayed for two days. To restore service, the *sshd* daemon would have to be killed and restarted.

[3]This binary also incorporated the backdoor patch, and was used for the backdoor experiment. The same synthetic normal profile was used for both experiments.

7.4.2 Linux capability exploit

The Linux capability attack takes advantage of a coding error in the Linux kernel 2.2.14 and 2.2.15. Within the Linux kernel, the privileges of the superuser are subdivided into specific classes of privileges known as *capabilities* which can be individually kept or dropped. This mechanism allows a privileged program to give up the ability to do certain actions while preserving the ability to do others. For example, a privileged program can choose to drop the capability that allows it to kill any process while keeping the capability to bind to privileged TCP/IP ports. In vulnerable versions of the Linux kernel, this capabilities code has a mistake that causes a process to retain capabilities that it had attempted to drop, even when the drop capability request returns with no errors.

A script published on BUGTRAQ [91] exploits this flaw by using *sendmail* [27], a standard program for transporting Internet email. It tells *sendmail* (version 8.9.3) to run a custom `sendmail.cf` configuration file that causes it to create a setuid-root shell binary in `/tmp`[4]. Normally, *sendmail* would drop its privileges before creating this file; with this kernel flaw, though, the drop privileges command malfunctions, allowing the shell binary to be given full superuser privileges.

Profiles based on normal *sendmail* usage on my home computers, lydia and atropos (my desktop and laptop, in June 2000), were used to test this exploit. The exploit script caused *sendmail* to behave very unusually, producing multiple anomalous processes, some with a LFC of 47 or more with responses disabled. This level of anomalous behavior may seem remarkable, but it is understandable when we realize that *sendmail*'s configuration language is very complex, and that the exploit uses a very unusual custom configuration file. In effect, this custom configuration turns *sendmail* into a file installation utility; thus, pH sees the reconfigured *sendmail* as a

[4]A setuid-root executable always runs as the superuser (root), no matter the privileges of the exec-ing process.

completely different program.

This behavior change causes pH to react vigorously. With *delay_factor* set to 4, the attack's processes were delayed for hours (before being manually killed) and were prevented from creating the setuid-root shell binary. Setting *abort_execve* to 1 made no difference, since the anomalously behaving *sendmail* processes did not make any execve calls.

pH 0.18 would react almost identically to this exploit. Even with *delay_factor* set to 1, the *sendmail* processes would be delayed for days (or much, much longer) before creating the setuid-root shell binary. Similarly, the *suspend_execve* setting would make no difference since this attack doesn't cause *sendmail* to make an execve call.

7.4.3 System-call Monitoring Experiments

Before pH, I tested the viability of system-call monitoring by monitoring system calls online and analyzing them offline. To further validate the design of pH, I revisited two past datasets to see how well pH would have performed under those circumstances. Using the *pH-tide* offline system-call analysis program included in the pH distribution, I analyzed the *named* [116] and *lpr* [49] datasets. It turns out that pH could detect both attacks; the response story, though, is a bit more complicated.

named buffer overflow

The Berkeley Internet Name Daemon (BIND) [26] is the reference implementation of the Domain Name Service (DNS), the standard Internet service that maps hostnames to IP addresses. *named* is the program in the BIND package that actually services DNS requests.

Several security vulnerabilities have been found in *named* over the years. In May 1998, ROTShB distributed an exploit script [96] that creates a remotely accessible superuser shell by overflowing a buffer in *named*'s inverse query routines. To test this exploit, I recorded *named*'s normal behavior on a secondary domain name server in the UNM Computer Science Department from June 14 through July 16, 1998 using a Linux kernel modified to log the system calls of specified binaries. During this month *named* made approximately nine million system calls. After manually annotating with parent/child fork information, re-analysis of this dataset using *pH-tide* and a window length of 9 produces a lookahead pair profile with 2137 pairs.

The exploit script was run twice: one where the *id* command was run, and one where the superuser shell was immediately exited. When these traces are compared against the *named* normal profile, the first exploit run produced 7 anomalies (LFC 7), while the second produced 5 anomalies (LFC 5). Delays would not have been sufficient to stop either attack; however, setting *suspend_execve* to 1 would have caused pH to delay the attacked *named* process for two days. Again, to regain service, *named* would have to be restarted.

lpr temp file exploit

lpr is a standard BSD UNIX program that sends data to a local or remote printer. In 1991 [8lgm] reported that the *lpr* program on SunOS 4.1.1 and other UNIX systems only used 1000 temporary filenames for pending jobs. By inserting symbolic links into *lpr*'s queue directory, an attacker could use *lpr* to overwrite arbitrary local files.

The [8lgm] *lprcp* attack script takes advantage of two infrequently used command line flags, `-q` and `-s`. The `-q` tells *lpr* to place the job in the queue instead of printing it, and the `-s` flag instructs *lpr* to create a symbolic link to `target` instead of copying it to the queue directory. The attack is very simple, and basically works as follows:

- Create a symbolic link to the target file: `lpr -q -s target`
- Increment the temporary file counter: `lpr /nofile` 999 times
- Print the new contents of the file: `lpr source`

When the last *lpr* command copies the source file to its queue directory, it follows the symbolic link in that directory to the desired target, overwriting it.

To see how well pH could distinguish this attack from normal *lpr* behavior, I installed the *lpr* binary from SunOS 4.1.1 and installed on SunOS 4.1.4 hosts at the MIT Artificial Intelligence Laboratory. I also installed a script which used *strace* to record *lpr*'s system calls. Over the course of two weeks, Febuary 18 through March 4, 1997, *lpr* was run 2766 times on 77 different hosts [49]. Analysis of these traces with *pH-tide* and a window size of 9 produced a lookahead pair profile with 2144 entries.

When compared with this normal profile by *pH-tide*, the anomalies of the 1001 attack *lpr* traces fall into three categories. The first trace (`lpr -q -s`) produces 4 anomalies (LFC 4), the middle 999 traces (`lpr /nofile`) each produce 2 anomalies (LFC 2), and the last trace (the final copy) generates 7 anomalies (LFC 7). In total, these traces produce 2009 anomalous system calls.

If pH were ported to SunOS 4.1.4, though, it would not have been able to stop this attack — even with all of these anomalies. With $delay\_factor$ set to 1, the first several *lpr* requests would each be delayed for a few seconds; however, once pH had recorded $anomaly\_limit$ anomalies, it would automatically tolerize the *lpr* profile, preventing pH from generating any further anomalies. Increasing $anomaly\_limit$ from 30 to 2500 would cause every job to be delayed; even this change, though, would only cause pH to delay the attack script for less than an hour. Setting $suspend\_execve$ to 1 would not change this result because *lpr* does not make any execve system calls

during this attack.

7.5 Intelligent Adversaries

The past sections have shown that pH can respond to many kinds of attacks. Each of these attacks, though, were developed on systems that did not monitor system call activity. Could attackers design their intrusion to evade detection by pH? To examine this possibility, consider one of the most challenging attack scenarios: a program containing trojan code. pH successfully stopped the *su* backdoor described in Section 7.3.2; what would it take to hide it from pH?

The first thing the attacker would have to do is to obtain the correct version of *su*. It turns out that many root kits contain utilities based on older versions or different code bases than current distributions. For example, one rootkit I recently found had programs such as login and netstat from 1994. Also, different distributions often use their own versions of basic commands such as *su* and *login*. Even if the binaries come from the same code base, they may be built with different options.

To see how this diversity could work to pH's advantage, I tested Red Hat 7.1's version of *su*. This version functions correctly on Debian systems even though it is based on a different (but related) code base. Running this *su* versus the normal profile generated by Debian's *su* listed in Table 7.7 produced a maximum LFC of 25 before the password prompt was printed. Such a concentration of anomalies would cause the program to be delayed for days even without the *suspend_execve* mechanism.

If we assume that the attacker has modified the correct program version and has tricked an administrator to install the modified binary on the target system (for example, through a compromised security update), then we are left with the following question: Could the attacker modify *su* in such a way that use of the backdoor was

invisible to pH?

The *su* backdoor described in Section 7.3.2 only produced one anomalous sequence:

fstat, mmap, mprotect, mmap, mmap, close, munmap, getpid, execve

Except for the $l = 4$ lookahead pair of (close, *, *, execve), all of the lookahead pairs encoded in this sequence are anomalous. To hide this sequence from pH, we have to make the execve look normal, meaning that it must occur in a context where execve's normally happen. This context must be true for every system call in pH's sequence window.

To make things simple, let us assume that we only have a window size of 2, and so we only have lookahead pairs for $l = 2$. Our anomaly, then, only consists of one lookahead pair: (getpid, execve). We now need to form a chain of system calls that can be inserted between getpid and execve which would mask our anomalous use of execve. The simplest scenario would be if *su*'s normal profile contained the lookahead pairs (getpid, **x**) and (**x**, execve). If this were the case, we could hide the backdoor's execve by preceding it with system call **x**.

Reality is not quite so simple. *su* has 111 $l = 2$ lookahead pairs in its normal profile out of 1377 total lookahead pairs. Of these 111, execve is the current system call in two of them: (chdir, execve) and (setuid, execve). Also, getpid is the position 1 system call in two other pairs: (getpid, rt_sigaction) and (getpid, brk). There is no overlap between these two sets, and so we must look for at least one other system call to connect them.

The close system call can serve this role: *su*'s normal profile has both the lookahead pairs (rt_sigaction, close) and (close, setuid), giving us the chain of (getpid, rt_sigaction), (rt_sigaction, close), (close, setuid), and (setuid, execve), or more sim-

ply getpid, rt_sigaction, close, setuid, execve. Thus, if the inserted backdoor first executed a rt_sigaction, close, and setuid system calls before making an execve, pH would detect no anomalous $l = 2$ lookahead pairs. If the pH's window size is 2, the backdoor could now be used without any interference.

If pH uses a larger window size, though, the execve will still appear to be anomalous. For example, the pair (getpid, *, *, *, execve) is not in *su*'s profile; instead, *su* only has two $l = 5$ lookahead pairs with execve as the current system call: (socketcall, *, *, *, execve) and (rt_sigaction, *, *, *, execve). If pH used a window size of 5, these lookahead pairs would also have to be masked with additional system calls.

This example shows that although possible in principle, it can be difficult in practice to create trojan code that is invisible to pH, even if we assume that the attacker knows the correct program version and the precise contents of a program's normal profile.

7.6 Summary

Table 7.9 summarizes pH's response to the security violations reported in this chapter. pH was able to successfully detect all of these attacks, and it was able to stop all except for the *lpr* temp file vulnerability. These results show that pH can successfully defend a monitored program against many kinds of attacks by observing unusual patterns in its behavior. pH detects this unusual behavior by observing novel system call patterns produced by previously unexecuted code paths. Because such code paths also correlate with other problems such as the misconfiguration of a network service, pH can also detect administration issues before they become otherwise apparent.

pH's responses can stop attacks that generate many anomalous system calls in one process or that make anomalous execve calls; pH is less able to stop attacks

Attack	Attack Type	Normal Type	Effective Response? Delay	Suspend	Both
inetd chargen	denial of service	real	yes	no	yes
fetchmail	buffer overflow	real	no	yes	yes
su back door	trojan	real	no	yes	yes
Linux ptrace/execve	kernel (race)	real	no	yes	yes
sshd overflow [106]	buffer overflow	synthetic	no	yes	yes
sshd back door [106]	trojan	synthetic	no	yes	yes
Linux capability [106]	kernel (error)	real	yes	no	yes
named (BIND) [117]	buffer overflow	real	no	yes	yes
lpr lprcp [49]	temp file	real (SunOS 4)	no	no	no

Table 7.9: A comparison of attack responses. The first five are discussed earlier in this chapter; the other five attacks were first presented in the cited papers. The first three of these attacks were tested with an earlier version of pH, while responses to the last two attacks were inferred based on offline data. Note that of these ten attacks, only two could not be stopped by pH's responses: the *inetd* failed fork and the *lpr* temp file attacks.

that only produce a few clustered anomalies per process. Most attacks that can be exploited remotely (such as the *sshd* and *fetchmail* attacks) fall into the former category; thus, pH is well-equipped to defend a host against attacks made by outside, unauthorized users.

Chapter 8

Discussion

The past several chapters documented the creation and testing of pH. This chapter reviews these results and places them in perspective. The first part describes the concrete contributions of the research and explains why these advances are important. Section 8.2 explains the limitations of the current pH prototype and suggests several ways in which it could enhanced. The last section places pH in the context of a full homeostatic operating system and describes my view of how pH-like mechanisms could make our computers more stable, secure, and autonomous.

8.1 Contributions

This work makes several contributions to the literature of computer security and operating systems. It has shown that system-call monitoring and response can be performed efficiently in real-time, can detect problems such as configuration errors, and can stop a variety of security violations. The rest of this section discusses past chapters and explains these contributions in more detail.

Before any new operating system mechanism can be widely deployed, it must be

shown to have minimal performance impact. The results in Section 5.10 show that lookahead-pair system call monitoring can be performed in real-time with little overhead: most system calls are slowed down by less than 2 μs, an X server experiences a 0.81% slowdown, and Linux kernel builds incur less than a 5% performance hit. pH's performance is competitive with other kernel security extensions such as Ko et al.'s security wrappers [60]. Most importantly, pH's overhead is small enough that normal users do not notice any difference in system performance.

Chapter 6 shows that the performance of both the sequence and lookahead pair methods are not especially sensitive to the choice of window size. Larger window sizes require more storage space, with the requirements growing linearly for lookahead pair method and exponentially for the full sequence method; the data requirements for convergence, however, grow sub-linearly with window size. The lookahead pair method is also shown to converge more quickly on normal program behavior than the sequences method. The modest storage requirements, extremely fast implementation, and the faster convergence properties of the lookahead pair method made it the natural choice for pH.

Chapter 6 then presents one of the principal contributions of this dissertation, showing that this monitoring can be combined with a few simple training heuristics and delay responses to form a system that can run on production systems with a relatively low rate of false positives. pH automatically obtains normal profiles for dozens of programs on a typical system. Most such profiles are of small utilities and scripts; given sufficient time and usage, however, pH also captures the normal behavior of large, complex programs like VMWare and the XFree86 X-Server. With aggressive parameter settings, pH requires user intervention once or twice a day; more conservative settings, however, require as few as one intervention every five days. Chapter 6 also showed that pH's profiles vary between programs and hosts. This diversity allows pH to provide customized protection for individual machines

and programs based on their usage patterns.

Chapter 7 shows that these normal profiles capture the normal code paths used by the monitored programs. Novel code paths are shown to correspond to interesting situations such as configuration errors, the use of (potentially dangerous) new program functionality, and the execution of foreign code. Although pH's detection of these phenomena is imperfect and could be subverted under certain conditions, in practice it is remarkably effective.

Chapter 7 also shows that execution delays are a safe, generic, and often effective response to anomalies. If there are only a few anomalies, and they are not execve calls, the delays are barely noticeable and at most contribute to the feeling that perhaps things aren't working quite right. In situations such as the use of the dangerous chargen service or the running of trojan code, delay can be sufficient to interrupt an attack before damage occurs. In addition, if a significant delay is imposed inappropriately, pH can be instructed to allow (tolerize) the suspect behavior. The program then continues to execute normally, and unless a timeout has been triggered, the program won't detect that anything has gone wrong.

Altogether, these results illustrate the viability of system-call monitoring and system-call delay response as a mechanism for improving system stability and security.

8.2 Limitations

Although pH is both remarkably efficient and effective, it is not perfect. It sometimes has difficulty capturing normal program behavior, it sometimes causes denials of service, and its current implementation is not easily portable from Linux. The following sections discuss these limitations and how they could be overcome.

8.2.1 Capturing Normal Behavior

Before an anomaly detection system can be effective it must have a model of normal behavior. pH requires that a profile be quiescent for a week before it is classified as normal. This constraint is rather stringent, and as a result, complicated programs are rarely monitored, and 70% or more of all executed programs are never monitored for anomalies.

One way to protect more programs would be to allow pH to detect anomalies while new pairs were still being added to a profile. For example, a per-profile anomaly threshold could be tightened as a profile stabilized. If pH never saw a locality frame count greater than 5 for a program for a week, it could then mount a response to any LFC greater than 5. Another approach would be to begin responses almost immediately upon observing program behavior. To keep the system running with acceptable performance, the delay equation could start off as being very close to zero. The equation could then be adjusted on a per-profile basis as each profile stabilizes.

A weakness of this strategy is that small changes in user behavior can result in very different patterns of system calls. In general the rate of novel sequences goes down; yet for all programs, there are discontinuities when usage patterns change. A profile that has "almost settled down" is not "almost stable"; the appearance of even a few novel sequences means that previously unseen code paths are being executed. The next new code path may generate a dozen new sequences or none at all.

Perhaps the simplest way to ensure that pH has profiles of normal program behavior would be for software developers to distribute default profiles of normal program behavior. These synthetic normal profiles could be easily generated by running some or all of a program's regression test suite. If pH detects anomalous program behavior relative to such a profile when a program is being used properly, then the program's test suite is not comprehensive enough. Over time, pH will replace many

of these profiles with ones that are specialized to the usage patterns of a host. These profiles would generally be smaller than the default synthetic normal profiles and would restrict program behavior to those code paths that are actually used on a given machine.

pH could also be improved through the addition of a userspace daemon to manage pH's profiles and regulate pH's responses. Such a daemon could note when new programs are run and use site-specific policies to determine whether it should be allowed or not. It could periodically scan the profiles directory for normal or almost-normal profiles that are likely to generate false positives. Except for the *tolerize_limit* mechanism, pH never forgets program behavior even if a given behavior was encountered only once. To mitigate this limitation, the daemon could prune profiles to remove entries that hadn't been recently used.

By correlating anomalies with network connections or log entries, a monitoring daemon could also decide whether a few scattered anomalies indicates that the system is under attack. It could then use this information to amplify pH's delay responses, or it could trigger a customized response. To prevent such a daemon from becoming a single source of failure, kernel-based mechanisms should continue to work on their own even in the absence of userspace analysis.

8.2.2 Denial of Service

By slowing down anomalously behaving programs, pH can prevent attackers from gaining unauthorized access; in the process, however, pH can also prevent legitimate access. The low false-positive rates reported in Chapter 6 show that pH rarely causes problems on normally behaving systems. What if an attacker deliberately attempts to provoke pH?

For example, it is possible for an attacker to cause a web server to behave un-

usually merely by sending it packets of random data. If pH had a normal profile for this server, this random data could cause pH to delay one or more of the server's processes. Further, if these anomalies were in the server's master process, pH could prevent other legitimate users from accessing web content.

This problem can be mitigated through a technique borrowed from the software rejuvenation field [20, 111]: a "watchdog" daemon could automatically kill and restart the web server whenever pH attempted to delay it. Such a daemon would only be a partial solution, because some connections could be refused while the server was restarting.

If a system is under vigorous attack (or is merely experiencing an unusually high load), it is possible for pH's responses to make a bad situation worse. To restore service, an administrator might have to tolerize the affected programs or disable pH's responses. Such actions would then leave the system open to attack.

One particularly effective attack would be for an attacker to use random but benign messages to trigger numerous false alarms. An administrator might then decide to turn off pH; once it was out of the way, the attacker could then exploit a real vulnerability and gain access without interference.

One of the strengths of pH, however, is its ability to adapt to new circumstances. Thus, if the administrator manually incorporated the benign behaviors into the attacked service's normal profile by tolerizing and normalizing it, pH could still defend the system against the real attack.

To summarize, pH's responses can cause denials of services that could be exploited by an attacker. A vigilant administrator, however, can use pH's adaptability to minimize these disruptions in service and still prevent surreptitious penetrations.

8.2.3 Portability

pH is implemented as an extension to the Linux 2.2 kernel running on x86-compatible processors. With a few changes, pH should also be able to run on other processors supported by the Linux kernel. Because most UNIX variants use monolithic kernels and support similar system call interfaces, it should be straightforward to port pH's core analysis routines to such systems. Because the data structures and functions that pH modifies differ significantly between Linux and other UNIX kernels, the interface portions of pH would be a bit more difficult; however, given source code access, it would be straightforward to port pH to FreeBSD, Solaris, HP-UX, or other UNIX systems.

pH captures the essence of a program's interactions with the outside world by observing its system calls. On systems that do not support a traditional system-call interface, pH would have to use other techniques to observe program behavior. Systems such as Exokernel [57] and L4 [66] have very small kernel interfaces and instead use inter-process communication to implement I/O operations such as file and network access, while systems like SPIN [11, 10] allow the creation of application-specific system calls. In both of these situations, multiple specialized interfaces replace UNIX system calls. To monitor similar kinds of program behavior, then, pH would have to monitor each of these interfaces. Because of the extreme performance constraints of the kernels in these systems, and because much of a UNIX kernel's functionality is often implemented in userspace processes, pH's behavior monitoring might appropriately be done in the userspace processes themselves. Thus, where on a UNIX system there is one component — the kernel — that must be modified to implement pH, these systems may require the modification of several components. Most of the time, the modularity of these operating systems is thought to make it easier to implement novel operating system mechanisms; this modularity, however, makes pH's type of global monitoring more difficult to implement.

Microsoft Windows 2000 and Windows XP [28] support kernel interfaces that are similar to UNIX system calls in size and functionality. It would be possible to port pH to Windows by having it monitor this interface; this ported pH, though, may not be as effective on Windows as it is on UNIX systems. Applications on UNIX are typically composed of multiple processes each running different executables; in contrast, Windows applications are usually composed of a single multi-threaded process running a large executable linked against many dynamic libraries. Because they have many normally used code paths, it is often difficult for pH to build a complete normal profile of large programs.

Instead of capturing the behavior of entire applications, a better approach may be to monitor the execution of application components. Stillerman et al. [107] have shown that sequences of CORBA method invocations can be used to detect security violations; in a similar manner, it should be possible to use DCOM method invocations to detect security violations in Windows applications. Because both DCOM and CORBA offer interposition mechanisms, a pH-like system for these object systems need not modify each component, although such modifications might be necessary for performance reasons.

For portability and security reasons, many newer applications target virtual machine environments such as the Java Virtual Machine [67] and Microsoft's .NET runtime environment. Inoue [51] has shown that method invocations can be efficiently monitored within a Java virtual machine environment; similar techniques should also be applicable to .NET. The increasing deployment of these technologies offers an opportunity to efficiently implement pH-like monitoring and response for large, distributed component-based applications.

pH can be most easily ported to systems that are similar to Linux. The more different the OS and application architectures are from UNIX processes, the more likely that an effective port will have to monitor and respond to program behavior in

different ways. As long as these ported systems follow the basic homeostatic pattern outlined in Chapter 3, however, they will be closely related to pH.

8.3 Computer Homeostasis

Although there are many ways pH could be extended and enhanced, it is important to understand that pH is perhaps best viewed as a low-level reflex mechanism, rather than as a full-fledged intrusion detection and response system. Just as our brain is not consulted when a hand gets too close to a hot stove, pH automatically acts to try and prevent damage to a system. Sometimes we may need to hold on to a hot handle to avoid dropping dinner on the floor; similarly, there are occasions when programs should behave unusually, such as when upgrading software or adding new peripherals. To complete the vision of pH, we need more connections between it and other parts of the system so that pH's responses may be better regulated.

As described in Chapter 3, living systems contain numerous, interconnected feedback loops that help maintain a stable internal state. These homeostatic mechanisms do not operate in isolation to fix problems; each does its part, but at the same time is regulated by other sensors and effectors. For example, humans shiver when it is cold and they have insufficient insulation. Humans also shiver when the immune system detects an infection and decides that a fever will help it defend the body.

In a similar fashion, a truly homeostatic operating system would integrate signals from log analyzers, network monitors, usage statistics, system call monitors, and other sources to detect and respond to unusual changes in system behavior. Further, the homeostatic organization described in Chapter 3 could be used to design novel mechanisms that would maintain other system properties such as system responsiveness and data integrity. To maintain those properties, it might make sense to look at other data streams such as keyboard events, window movements, or filesystem

operations. A sliding window approach can be used for internal interfaces such as library APIs or object methods; other data streams, though, may need other types of detectors.

The analysis of such mechanisms would preferably be distributed and loosely coupled, much as is done in the human immune system and in robotic subsumption architecture. For example, when a log analyzer detects unusual activity, it might start normal monitoring on certain programs and increase pH's $delay\_factor$. A monitoring daemon, then, might note that the log analyzer's anomalies are actually normal at this time of the month, and so would reduce $delay\_factor$.

This type of interplay between positive and negative reinforcement allows the immune system to make subtle decisions without depending on a vulnerable central controller. By carefully connecting systems that are individually robust and at least sometimes useful, it should be possible to create an artificial system that is globally robust, accurate, redundant, and hard to subvert. Such a system may be developed incrementally, by demonstrating the utility of each component independently, and then testing the integrated system under realistic conditions. The resulting system may not be easy to understand and may sometimes have unexpected, and even pathological behavior; the reward for this effort will be systems that degrade gracefully in response to error and attack and that are capable of responding to situations beyond the scope of their original design.

The true purpose and result of this work, then, has been to show that reflex-like behavior-based mechanisms can improve the stability and security of a conventional operating system. The promise of a complete homeostatic operating system, though, requires that pH be integrated with other behavior-based and knowledge-based monitoring and responses systems. Building this larger system will be challenging and will take many years. I believe the rewards will be worth the effort.

References

[1] David H. Ackley. ccr: A network of worlds for research. In C.G. Langton and K. Shimohara, editors, *Artificial Life V*, pages 116–123. MIT Press, 1997.

[2] Thayne Allen. LIDS — deploying enhanced kernel security in linux. http://rr.sans.org/linux/lids.php, February 12, 2001.

[3] James P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, 1980.

[4] antirez. Fetchmail security advisory. BUGTRAQ Mailing list (bugtraq@securityfocus.com), August 10, 2001. Message-ID: <20010810000341.C1176@blu>.

[5] Ivan Arce. SSH-1.2.27 & RSAREF2 exploit. BUGTRAQ Mailing list (bugtraq@securityfocus.com), December 14, 1999. Message-ID: <3856C3EF.230F0AE@core-sdi.com>.

[6] Trustix AS. Trustix secure linux. http://www.trustix.net, January 2002.

[7] Stefan Axelsson. Intrusion detection systems: A taxomomy and survey. Technical Report 99-15, Dept. of Computer Engineering, Chalmers University of Technology, March 2000.

[8] Rebecca Gurley Bace. *Intrusion Detection*. Macmillan Technical Publishing, 1999.

[9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[10] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda D. McNamee, P. Pardyak, S. Savage, and E.G. Sirer. Spin — an extensible microkernel for application-specific operating system services. *Operating Systems Review*, 29(1):74–77, January 1995.

[11] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, Copper Mountain, CO, 1995.

[12] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. In *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX 2000)*, volume 2, January 25–27, 2000.

[13] Rodney A. Brooks. A robust layered control system for a mobile robot. A.I. Memo 864, Massachusetts Institute of Technology, September 1985.

[14] Rodney A. Brooks and Anita M. Flynn. Fast, cheap, and out of control: a robot invasion of the solar system. *Journal of The British Interplanetary Society*, 42:478–485, 1989.

[15] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, June 1997.

[16] Mark Burgess. cfengine home page. http://www.cfengine.org.

[17] Mark Burgess. Automated system administration with feedback regulation. *Software — Practice and Experience*, 28(14):1519–1530, December 1998.

[18] Mark Burgess. Computer immunology. In *Proceedings of the 12th system administration conference (LISA 1998)*, October 28, 1998.

[19] Michael G. Burke, Jong-Deok Choi, Stephen J. Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño dynamic optimizing compiler for java. In *Java Grande*, pages 129–141, 1999.

[20] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM Journal of Research & Development*, 45(2), March 2001.

[21] Jr. Charles A. Janeway and Paul Travers. *Immunobiology: the Immune System in Health and Disease*. Garland Publishing Inc., New York, second edition edition, 1996.

[22] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security*. Addison-Wesley Pub Co., 1994.

[23] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[24] Cisco Systems, Inc. Cisco secure intrusion detection system. http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/, 2000.

[25] Fred Cohen. The deception toolkit. http://www.all.net/dtk/, January 2002.

[26] Internet Software Consortium. Berkeley internet name daemon. http://www.isc.org, 2002.

[27] Sendmail Consortium. sendmail.org. http://www.sendmail.org/, 2000.

[28] Microsoft Corporation. Microsoft home page. http://www.microsoft.com.

[29] Microsoft Corporation. Repairing office installations. http://www.microsoft.com/office/ork/xp/two/adma01.htm, April 4, 2001.

[30] Transmeta Corporation. Crusoe processor: Longrun technology. http://www.transmeta.com /crusoe/lowpower/longrun.html, January 2000.

[31] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious server security. In *14th USENIX Systems Administration Conference (LISA 2000)*, New Orleans, LA, December 2000.

[32] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[33] Helena Curtis and N. Sue Barnes. *Biology*. Worth Publishers, Inc., New York, 5th edition, 1989.

[34] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, April 23, 1999.

[35] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[36] Renaud Deraison et al. The nessus project. http://www.nessus.org, March 2002.

[37] Solar Designer. Linux kernel patch from the openwall project. http://www.openwall.com/linux/, 2001.

[38] Sebastian Elbaum and John C. Muson. Intrusion detection through dynamic software measurement. In *Proceedings of the 1st Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 9–12, 1999. The USENIX Association.

[39] D. Endler. Intrusion detection: Applying machine learning to solaris audit data. In *Proceedings of the 1998 Annual Computer Security Applications Conference (ACSAC'98)*, pages 268–279, Scottsdale, AZ, December 1998. IEEE Computer Society Press.

[40] Yasuhiro Endo, James Gwertzman, Margo Seltzer, Christopher Small, Keith A. Smith, and Diane Tang. VINO: The 1994 fall harvest. Technical Report TR-34-94, Harvard Computer Center for Research in Computing Technology, 1994.

[41] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[42] Dan Farmer and Wietse Venema. Satan home page. http://www.fish.com/satan/, 1995.

[43] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[44] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[45] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 9–12, 1999. The USENIX Association.

[46] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[47] L.T. Heberlein, G.V. Dias, K.N. Levitt, and B. Mukherjee. A network security monitor. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 296–304, 1990.

[48] G.J. Henry. The fair share scheduler. *Bell Systems Technical Journal*, 63(8):1845–1857, October 1984.

[49] S. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[50] Steven A. Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[51] Hajime Inoue and Stephanie Forrest. Generic application intrusion detection. Technical Report TR-CS-2002-07, University of New Mexico, 2002.

[52] Internet Security Systems, Inc. RealSecure. http://www.iss.net/securing_e-business/security_products/intrusion_detection/index.php, 2001.

[53] Internet Security Systems, Inc. RealSecure OS Sensor. http://www.iss.net/securing_e-business/security_products/intrusion_detection/realsecure_ossensor/, 2001.

[54] M. Szychowiak J. Brzeziński. Self-stabilization in distributed systems — a short survey. *Foundations of Computing and Decision Sciences*, 25(1), 2000.

[55] Anita Jones and Song Li. Temporal signatures for intrusion detection. In *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 10–14, 2001.

[56] Anita Jones and Yu Lin. Application intrusion detection using language library calls. In *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 10–14, 2001.

[57] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.

[58] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[59] Gene H. Kim and Eugene H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. Technical Report CSD-TR-94-012, Department of Computer Sciences, Purdue University, February 21, 1994.

[60] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 14–17, 2000.

[61] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in priviledged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 5–9, 1994.

[62] David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. AAAI Press/The MIT Press, 1998.

[63] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, September-October 1997.

[64] Benjamin A. Kuperman and Eugene Spafford. Generation of application level audit data via library interposition. Technical Report CERIAS TR 99-11, COAST Laboratory, Purdue University, West Lafayette, IN, October 1999.

[65] Wenke Lee, Salvatore Stolfo, and Patrick Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI methods in Fraud and risk management*, 1997.

[66] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[67] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley Longman, Inc., 2nd edition, April 1999.

[68] Ulf Lindqvist and Phillip A. Porras. eXpert-BSM: A host-based intrusion detection solution for Sun Solaris. In *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 10–14, 2001.

[69] Tom Liston. Welcome to my tarpit: The tactical and strategic use of LaBrea. http://www.hackbusters.net/LaBrea/LaBrea.txt, January 2002.

[70] Peng Liu. DAIS: A real-time data attack isolation system for commercial database applications. In *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 10–14, 2001.

[71] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Neumann, H.S. Javitz, A. Valdes, and T.D. Garvey. A real-time intrusion detection expert system (IDES) — final technical report. Computer Science Laboratory, SRI International, Menlo Park, California, February 1992.

[72] Michael R. Lyu, editor. *Software Fault Tolerance*, volume 3 of *Trends in Software*. John Wiley & Sons, New York, 1995.

[73] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in Phalanx. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 51–58, 1998.

[74] Carla Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the New Security Paradigms Workshop 2000*, Cork, Ireland, Sept. 19–21, 2000. Association for Computing Machinery.

[75] Henry Massalin and Calton Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, 1989. Revised March 1990.

[76] Roy A. Maxion and Kymie M. C. Tan. Benchmarking anomaly-based detection systems. In *International Conference on Dependable Systems and Networks*, pages 623–30, New York, NY, June 25–28, 2000. IEEE Computer Society Press.

[77] Roy A. Maxion and Kymie M. C. Tan. Anomaly detection in embedded systems. *IEEE Transactions on Computers*, 51(2):108–120, February 2002.

[78] C.C. Michael and Anup Ghosh. Two state-based approaches to program-based anomaly detection. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00)*, New Orleans, LA, December 11–15, 2000.

[79] Sun Microsystems. The java hotspottm virtual machine. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html, 2001. White Paper.

[80] John C. Munson and Scott Wimer. Watcher: The missing piece of the security puzzle. In *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 10–14, 2001.

[81] National Computer Security Center. Trusted product evaluation program (TPEP) evaluated products by rating. http://www.radium.ncsc.mil/tpep/epl/epl-by-class.html, January 2001.

[82] National Security Agency. Security-enhanced linux. http://www.nsa.gov/selinux/, January 2002.

[83] Ruth Nelson. Unhelpfulness as a security policy or it's about time. In *Proceedings of the 1995 New Security Paradigms Workshop*, La Jolla, CA, August 22–25, 1995. IEEE Press.

[84] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, volume DOD 5200.28-STD (The Orange Book). Department of Defense, 1985.

[85] Paolo Perego and Aldo Scaccabarozzi. AngeL — the power to protect. http://www.sikurezza.org/angel/, January 2002.

[86] P. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the National Information Systems Security Conference*, 1997.

[87] J. Postel. Request for comment (RFC) 864: Character generator protocol, May 1983.

[88] J. Postel. Request for comment (RFC) 867: Daytime protocol, May 1983.

[89] Psionic Software. Logcheck version 1.1.1. http://www.psionic.com/abacus/logcheck, January 2001.

[90] Inc. Psionic Technologies. Psionic portsentry. http://www.psionic.com/products/portsentry.html, March 2002.

[91] Wojciech Purczynski. Sendmail & procmail local root exploits on Linux kernel up to 2.2.16pre5. BUGTRAQ Mailing list (bugtraq@securityfocus.com), June 9, 2000. Message-ID: <Pine.LNX.4.21.0006090852340.3475-300000@alfa.elzabsoft.pl>.

[92] Wojciech Purczynski. ptrace/execve race condition exploit (non brute-force). BUGTRAQ Mailing list (bugtraq@securityfocus.com), March 27, 2001. Message-ID: <Pine.LNX.4.30.0103271358190.31983-200000@alfa.elzabsoft.pl>.

[93] D. J. Ragsdale, C. A. Carver, J. W. Humphries, and U. W. Pooch. Adaptation techniques for intrusion detection and intrusion response systems. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 2344–2349, Nashville, Tennessee, October 8–11, 2000.

[94] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accounted execution of untrusted code. In *IEEE Hot Topics in Operating Systems (HotOS) VII*, March 1999.

[95] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, September/October 2001.

[96] riders of the short bus (ROTShB). named warez. BUGTRAQ Mailing list (bugtraq@securityfocus.com), May 31, 1998. Message-ID: <199805310638.CAA18523@netspace.org>.

[97] SSH Communications Security. SSH secure shell. http://www.ssh.com/products/ssh/, 2000.

[98] LEE A. SEGEL and IRUN R. COHEN, editors. *Design Principles for the Immune System and Other Distributed Autonomous Systems*, chapter Introduction to the Immune System, pages 3–28. Oxford University Press, 2001. by Steven A. Hofmeyr.

[99] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[100] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*. The USENIX Association, April 1999.

[101] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proceedings of the National Information Systems Security Conference*, 1998.

[102] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, Chatham, MA, May 1997.

[103] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, January 1996.

[104] Michael D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, Boston, MA, January 18, 2000. Invited Lecture.

[105] Software Systems International. Cylant division home page. http://www.cylant.com, January 2001.

[106] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 14–17, 2000.

[107] Matthew Stillerman, Carla Marceau, and Maureen Stillman. Intrusion detection for distributed applications. *Communications of the ACM*, 42(7):62–69, July 1999.

[108] Symantec. Norton antivirus 2002. http://www.symantec.com/nav/nav_9xnt/, 2001.

[109] The Honeynet Project. The honeynet project home page. http://project.honeynet.org/, January 2002.

[110] timecop. Root kit SSH 5.0. http://www.ne.jp /asahi/linux/timecop/, January 2000.

[111] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):62–71, June 2001.

[112] Arthur J. Vander, James H. Sherman, and Dorothy S. Luciano. *Human Physiology: the Mechanisms of Body Function*. McGraw-Hill Publishing Co., New York, 1990.

[113] Wietse Venema. TCP WRAPPER: network monitoring, access control, and booby traps. In *Proceedings of the 3rd UNIX Security Symposium*, 1992.

[114] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[115] X. Wang, D. Reeves, S.F. Wu, and J. Yuill. Sleepy watermark tracing: an active network-based intrusion response framework. In *Proceedings of the IFIP Conference on Security*, Paris, 2001.

[116] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, CA, 1999. IEEE Computer Society.

[117] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[118] Stephen Young. Dr. Stephen Young's home page. http://rheumb.bham.ac.uk/youngsp.html, February 1995.

[119] Diego Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, August 2001.

[120] Marek Zelem, Milan Pikula, and Martin Ockajak. Medusa DS9 security system. http://medusa.fornax.sk, January 21, 2001.

[121] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 15–26, October 1997.

[122] D. Zimmerman. Request for comment (RFC) 1288: The finger user information protocol, December 1991.